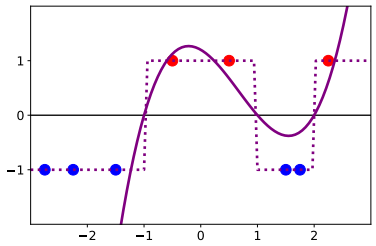
CHAPTER

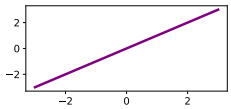
6 Neural Networks

We have seen that linear classifiers can be made more powerful via feature transforms. For example, creating a collection of nonlinear features, such as polynomial transforms, can be used to correctly classify data sets that would otherwise not be separable. However, selecting a "good" set of features often requires some domain knowledge. **Neural networks** instead define a more complex, but fully parameterized nonlinear function that can be jointly optimized. Intuitively, we can view this construction as a type of "trainable" feature transform.

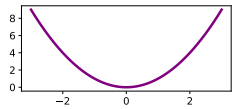
Example 6-1 : Non-linearly separable data

In Chapter 4, we saw that data sets that were not linearly separable could often be separated using a perceptron defined on a higher-dimensional transformation of the features, such as polynomial transforms:

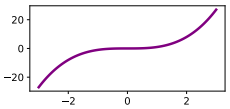




$\phi_1(x) = x$



$\phi_2(x) = x^2$

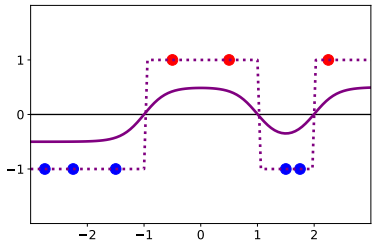


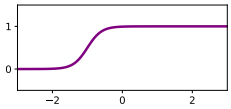
$\phi_3(x) = x^3$

$$r(x) = \beta_0 + \beta_1\,\phi_1(x) + \beta_2\,\phi_2(x) + \beta_3\,\phi_3(x)$$

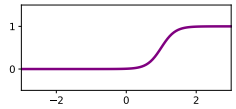
Although no linear function of the scalar feature x can correctly separate the training data, a cubic function of x (or equivalently, a linear function of degree-3 polynomial features of x) can.

Many other features can also separate these data – for example, we can use features defined by "soft step" functions (here, the logistic σ). Then, a linear combination of the transformed features ϕ can be thresholded to give the desired decision function:

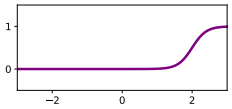




$\phi_1(x) = \sigma(5x + 5)$



$\phi_2(x) = \sigma(5x - 5)$



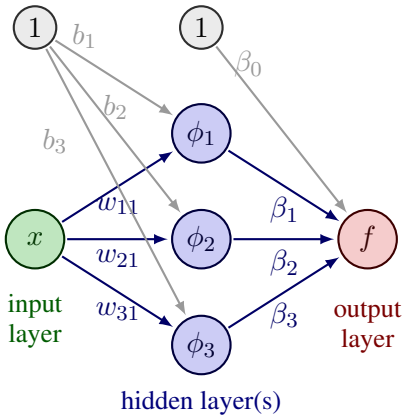
$\phi_3(x) = \sigma(5x - 10)$

$$r(x) = \beta_0 + \beta_1\,\phi_1(x) + \beta_2\,\phi_2(x) + \beta_3\,\phi_3(x)$$

Although the "soft step" features of Example 6-1 may not seem as general-purpose as, say, polynomial features, they are appealing in a different way: each feature $\phi_i(x)$ is itself a (smoothed) perceptron function. Thus, the overall (smoothed) output is,

$$f(x) = \sigma(r(x)) = \sigma(\ \beta_0 + \beta_1\,\sigma(w_{11}x + b_1) + \beta_2\,\sigma(w_{21}x + \beta_2) + \beta_3\,\sigma(w_{31}x + b_3)\ ),$$

i.e., a perceptron whose inputs are also perceptrons, or **multi-layer perceptron**, an alternative name for a neural network. Thus our model consists of a nonlinear function $f(x\,;\,\theta)$ whose parameters, θ consist of the complete set of coefficients, $\theta = [b_1, w_{11}, b_2, \ldots, \beta_2, \beta_3]$. Moreover, the function f is differentiable, allowing us to apply gradient based optimization to simultaneously learn a useful set of features ϕ_i (defined by the parameters corresponding to $\{b_i, w_{i1}\}$) along with their coefficients $\{\beta_0, \ldots, \beta_3\}$.



We can visualize the structure of this computation graphically, by associating nodes in the graph with features and intermediate quantities, and (directed) edges associated with each weight (parameter) that multiplies its associated input. We organize this structure in terms of *layers*, with the input feature(s) x forming the **input layer**, the output(s) of the function, f, forming the **output** layer, and the intermediate quantities, the features ϕ_i, called a **hidden layer** (since they are neither input nor output, but a computation used internally). Although shown explicitly here, we typically do not bother drawing the constant features or their associated bias weights (here, b_i and β_0, shown in gray). We call this structure a *two-layer* neural network – one hidden layer, and one output layer (we do not count the input layer).

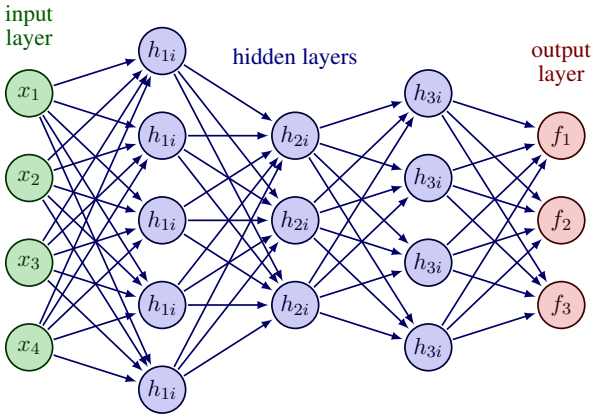
In defining our features ϕ_i and our output f, we have used the logistic sigmoid function $\sigma(r)$, to emphasize the similarity of these steps to a standard linear classifier, in which we used the logistic transform to define a smooth surrogate loss. Intuitively, when the linear response $r_i = b_i + w_{i1}x$ is large and negative, we have $\Phi_i(x) \to 0$; when r_i becomes large and positive, we have $\Phi_i(x) \to 1$, so that the feature is "activated". For instance, in Example 6-1, Φ_1 became activated when $x > -1$, while Φ_2 became activated when $x > +1$, and Φ_3 only when $x > 2$. For this reason, the nonlinear transformation of r_i is often called the **activation function**. However, in many multi-layer perceptrons it may be convenient to adopt different types of activation function instead.

6.1 Feed-forward Neural Networks

In order to discuss a general neural network model structure, we require some notation. Suppose that our neural network consists of L layers (so, $L-1$ hidden layers and one output), of sizes H_l, $l \in \{1, \ldots L\}$, respectively. Each hidden node h_{lj}, indicating the jth node in layer l, is computed by taking the weighted sum of the previous layer's hidden nodes, and applying an activation function $a_l(\cdot)$:

$$r_{lj} = \sum_i w_{lij} h_{(l-1)i} + b_{lj} \qquad\qquad h_{lj} = a_l(\ r_{lj}\ ) \tag{6.1}$$

where w_{lij} represents the weight of node i at layer $l-1$ in calculating node j's response. For convenience, we define $h_{0i} = x_i$ and $h_{Lk} = f_k$ so that (6.1) can be applied to define the first hidden layer in terms of the inputs, and the output layer in terms of the last hidden layer, $L-1$. Neural networks of this form are sometimes called **feed-forward neural networks**, since the calculation of each layer depends on the preceding layer's outputs.



Network Sizes

number of hidden nodes and layers influences flexibilty of the function

how many parameters?

Example: 2-layer 2D data, $L_1 = ...$

Universal approximation property

Backpropagation

A powerful feature of neural networks is that, although they can define an extremely flexible class of functions, they can also be trained in a straightforward manner using gradient descent. This is remarkably easy to do, thanks to the chain rule of derivatives.

Generic chain rule

We can think of the feed-forward calculation of the neural network as a "schematic" made up of various smaller functions – the computation of the intermediate quantities h_{lj} – each of which depends on the computation of previous quantities, and so on. Let us consider a tiny, abstract version of this process, in which we have a loss J, which depends on the true class and the output of our model, f; f depends on, say, two intermediate quantities, h and g, both of which depend on some parameter θ. Suppose that our current value of θ is $\theta = \theta_0$. Then, to evaluate the loss $J(\theta_0)$, we simply compute $h_0 = h(\theta_0)$ and $g_0 = g(\theta_0)$ at the current parameter value θ_0, then compute $f_0 = f(h_0, g_0)$, then $J(f_0)$. We call this evaluation process the **forward pass**.

add diagrams

Now, suppose we would like to evaluate the derivative of our loss, J, with respect to the parameter θ, at its current value θ_0, which we write as $\frac{\partial J}{\partial \theta}|_{\theta_0}$. To do so, we work backwards using the chain rule. We have,

$$\frac{\partial J}{\partial \theta}|_{\theta_0} = \frac{\partial J}{\partial f}|_{f_0} \frac{\partial f}{\partial \theta}|_{\theta_0}$$
$$\frac{\partial f}{\partial \theta}|_{\theta_0} = \frac{\partial f}{\partial g}|_{g_0} \frac{\partial g}{\partial \theta}|_{\theta_0} + \frac{\partial f}{\partial h}|_{h_0} \frac{\partial h}{\partial \theta}|_{\theta_0}.$$

In other words, the overall derivative of J is computed in terms of the values computed during the forward pass, combined with "local" derivatives of each component (J, f, g, h) with respect to their immediate arguments. We can view the computation as a **backward pass**: we first compute $\frac{\partial J}{\partial f}$ at f_0 and send this information "back" to f's node. At f's node, we computing f's derivative with respect to its arguments,

$\partial f/\partial g$ and $\partial f/\partial h$ and send $(\frac{\partial J}{\partial f})(\frac{\partial f}{\partial g})$ backwards to g, and $(\frac{\partial J}{\partial f})(\frac{\partial f}{\partial h})$ backwards to h. At g, we evaluate $\partial g/\partial\theta$ and send $(\frac{\partial J}{\partial f})(\frac{\partial f}{\partial g})(\frac{\partial g}{\partial \theta})$ to θ's node, and do the same from h's node. Finally, θ's node recieves two derivative messages, from g and h, and simply adds them together.

Computing gradients with backpropagation

Let us see how this works out in practice, on a neural network. Consider a simple, two-layer (one hidden layer, one output layer) neural network, with first layer weights w_{ij} and activation $a_1(\cdot)$, and output layer weights β_{jk} and activation $a_2(\cdot)$. Let x be a single data point; our first step is to compute the prediction at x, $f(x)$, and its contribution to the loss J, by performing a forward pass:

$$\begin{aligned} r_j &= \sum_i w_{ij}\, x_i + w_{0j} && \Rightarrow && h_j = a_1(r_j) \\ s_k &= \sum_j \beta_{jk}\, h_j + \beta_{0k} && \Rightarrow && f_k = a_2(s_k) \\ & && \Rightarrow && J = \sum_k (y_k - f_k)^2 \end{aligned}$$

where, for concreteness, we have assumed a squared error loss between y and f.

Now, our complete parameters θ can be grouped in two parts: the first layer, W, and the output layer, β. The gradient vector, $\nabla\theta$, consists of the derivatives with respect to each of these parameters. Let us first consider one of the parameters in the output layer, β_{jk}.

Given the values of the hidden nodes $\{h_j\}$, the output layer is simply a perceptron model, with surrogate loss J, using features $h_j(x)$. Thus, the gradient is exactly the same as computed previously:

$$\begin{aligned} \frac{\partial J}{\partial \beta_{jk}} &= \sum_{k'} \Big(\frac{\partial J}{\partial f_{k'}}\Big) \Big(\frac{\partial f_{k'}}{\partial r_{k'}}\Big) \Big(\frac{\partial r_{k'}}{\partial \beta_{jk}}\Big) \\ &= \Big((-2)(y_{k'} - f_{k'})\Big) \Big(a_2'(s_k)\Big) \Big(h_j\Big) \end{aligned}$$

where, in the last equality, we have used the fact that only output node k actually depends on the parameter β_{jk}; the rest have $\frac{\partial s_{k'}}{\partial \beta_{jk}} = 0$, for $k' \neq k$. The quantity $a_2'(s_k)$ is simply the slope of a_2 at the point s_k; for example, recall that for the logistic activation, $\sigma(s)$, we have $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. For $j = 0$ (a bias term), we simply take $h_0 = 1$.

Now, let us compute the elements of the gradient correspond to the weights w_{ij}. We have,

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k'} \Big(\frac{\partial J}{\partial f_{k'}}\Big) \Big(\frac{\partial f_{k'}}{\partial r_{k'}}\Big) \Big(\frac{\partial r_{k'}}{\partial w_{ij}}\Big)$$

but now, $r_{k'} = \sum_j \beta_{jk'} h_j$ depends on w_{ij} through the hidden node's output, h_j:

$$\begin{aligned} \frac{\partial r_{k'}}{\partial w_{ij}} &= \sum_{j'} \Big(\frac{\partial r_{k'}}{\partial h_{j'}}\Big) \Big(\frac{\partial h_{j'}}{\partial s_{j'}}\Big) \Big(\frac{\partial s_{j'}}{\partial w_{ij}}\Big) \\ &= \Big(\beta_j\Big) \Big(a_1'(s_j)\Big) \Big(x_i\Big) \end{aligned}$$

where again we use the fact that only s_j depends on w_{ij}; the derivatives of $s_{j'}$ are zero, for $j' \neq j$. This gives us the w elements of the gradient:

$$\frac{\partial J}{\partial w_{ij}} = \Big((-2)(y_{k'} - f_{k'})\Big) \Big(a_2'(s_k)\Big) \Big(\beta_j\Big) \Big(a_1'(s_j)\Big) \Big(x_i\Big).$$

Again, for bias terms (w_{0j}) we define $x_0 = 1$.

While the resulting gradient looks quite complicated, it is easily computed in a back-to-front pass. We evaluate the derivative of J with respect to each of the outputs, f_k, and pass these back to each output. Then, each f_k computes its slope a_2' at the value s_k (computed and saved in the forward pass), and passes the product of the derivative it received from its output, times the slope, times h_j, to parameter β_{jk}, and the same product except times β_{jk}, to node h_j.

add diagram

Node h_j recieves derivatives from each output f_k – it sums these together, multiplies by a_1' at value r_j (saved in the forward pass), and passes back the product times x_i to parameter w_{ij}. (For completeness, one could pass the product times w_{ij} to x_i, but this quantity is a constant.)

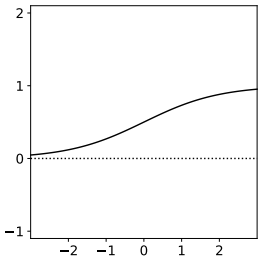
Thus, each node is performing a very simple operation: having saved its input values during the forward pass, it computes the derivative of its output with respect to each input, and passes this back to its respective input, multiplied by the sum of derivatives recieved from its outputs. After a backward pass, each parameter has been passed its component of the gradient, and we can perform an update. For batch or mini-batch gradient descent, we simply accumulate the gradient terms over the mini-batch until we are ready to take a step.

Activation Functions

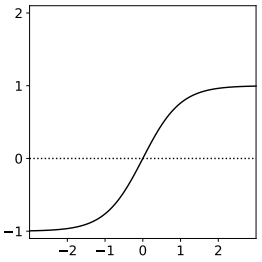
The activation function, $a_l(\cdot)$, of each layer l is an important component of the neural network architecture. Since we plan to train our neural network via gradient descent, it is important that the nonlinearities be differentiable, so that we can compute gradients.

There are two commonly popular types of activation function. The first are saturating, sigmoidal functions, such as the logistic sigmoid $\sigma(\cdot)$, whose output lies in the range $[0, 1]$. Another common variant of this type is the hyperbolic tangent function, $\tanh(\cdot)$, which forms a simlar, S-shaped curve whose output is bounded in $[-1, 1]$.

Saturating activation functions:



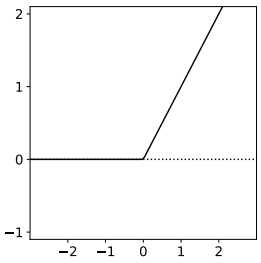
$a(r) = \frac{1}{1+\exp(-r)}$
Logistic sigmoid



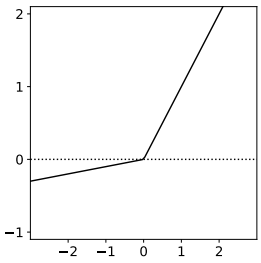
$a(r) = \frac{1-\exp(-2r)}{1+\exp(-2r)}$
Hyperbolic tangent

The other common category of activation functions are piecewise linear functions, the most common of which is the "rectified linear unit" or **ReLU** function. The ReLU activation function $a(r)$ simply returns its argument r if $r \geq 0$, and zero otherwise, so that $a(r) \in [0, \infty]$. However, while this activation function is easily differentiable (except at $r = 0$), its derivative is zero for data whose response r at that node is negative.

Piecewise linear activation functions:



$a(r) = \max[r, 0]$
Rectified Linear (ReLU)



$a(r) = \max[r, \epsilon\, r]$
Leaky ReLU

To see why this is a problem, imagine that our weights, w_{lij}, were all initialized to be negative. The first layer of hidden nodes, h_{1i}, are all positive (thanks to the form of ReLU), which means that the next layer's linear sums, r_{2j}, are all negative, so that $h_{2j} = 0$ for all j. Not only is our output zero, but its gradient is also zero, since any slight change to the w_{lij} will leave them negative and produce the same output. Our model is stuck at this poor setting. To avoid zero gradients, the **leaky ReLU** activation

follows a similar form, but uses a line with small but non-zero slope ϵ for negative arguments. (The parameter ϵ then becomes part of the architecture design.)

Output layer activation functions. Although it is possible to use different activation function at any layers of the neural network, it is very common to make them the same at all hidden layers. However, the output layer is often chosen differently.

For regression problems, we may not want to bound the values of our outputs to be in a pre-specified range ($[0, 1]$ for logistic, $[0, \infty]$ for ReLU, etc.). In such cases, it is common to make the output layer a linear function of the last hidden layer. By simply choosing the last activation function a_L to be linear, $a_L(r) = r$, our output layer takes the form of a linear regression model whose features are defined by the network.

For classification problems, we typically set the number of output nodes equal to the number of classes C. As in multi-class classification, this associates an output f_c with each possible class c; to predict a discrete class value, we choose the index c with the highest output. A common output layer activation in this setting is the **softmax**, or multilogit function, which operates on the vector of linear values, $\underline{r}_L = [r_{L1}, \ldots, r_{LK}]$ and outputs a vector of values $\underline{f}$:

$$\underline{f} = [f_1, \ldots, f_K] = softmax(\underline{r} = [r_1, \ldots, r_K]) \qquad f_c = \frac{\exp(r_c)}{\sum_k \exp(r_k)}$$

This ensures that the set of outputs, $\underline{f}$, are all positive and sum to one. We can interpret each output $f_c(x)$ as the model's estimate of the conditional probability, $p(Y = c|X = x)$, and apply the multi-class form of the log-loss (negative log likelihood),

$$J_{\text{NLL}} = -\sum_k \mathbb{1}[y = k] \log(\, f_k(x)\,))$$

i.e., the negative log of the model's estimated probability of the true class, y.

Practical considerations. Historically, saturating activation functions such as the logistic or hyperbolic tangent functions were most common in neural networks. However, with modern, large networks there are practical reasons to use ReLU or leaky ReLU activation functions. Evaluating the activation function itself, and its derivative, is much faster than saturating activations: ReLU involves a simple positivity check (its value is r and derivative 1 if $r > 0$, and otherwise both are zero). In contrast, the logistic function requires exponentiating r, the cost of which is hardware-dependent, but requires on the order of 20-40 floating point operations – so, much slower.

Obviously, however, the form of $a(\cdot)$ also affects the form of the possible functions f. However, as the number of layers and hidden nodes grows, we can hope that any reasonable choice of activation function will come "close enough" to the optimal predictor to be acceptable.

Example With ReLU activations, the outputs of the first layer are piecewise linear functions of x. Then, the outputs of the next layer are piecewise linear functions of the first layer – so, piecewise linear functions of x as well, although with potentially many more pieces. Thus, the final decision boundary is also piecewise linear. We can contrast this with the decision boundary of a logistic activation function, which is more smooth.

Form of the activation function can strongly influence form that f can take; e.g., ReLU-based networks are piecewise linear decision functions. Example.

6.2 Special Architectures

Residual Networks

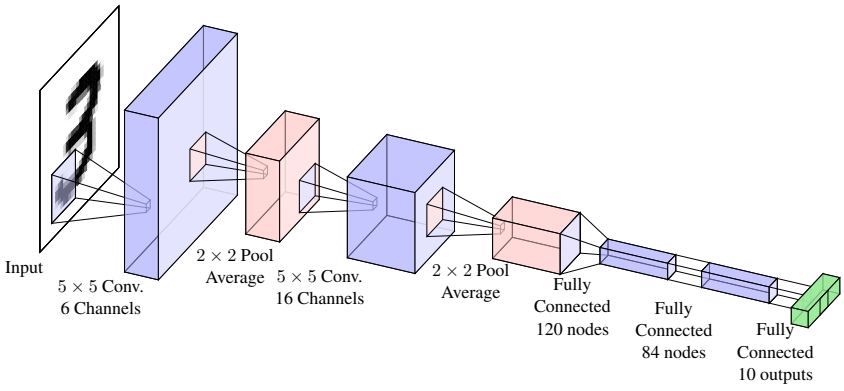
Convolutional Neural Networks

add Input image; index features as $x_{(ij)}$ for the (i, j)th pixel. Define a two-dimensional collection of hidden nodes, which all share a small, $k \times k$ collection of weights (called a "filter"); we define

$$h_{(ij)} = \sum_{k \in [-K,K]} \sum_{l \in [-L,L]} w_{k,l} x_{(i+k,j+l)}$$

In other words, the features used for $h_{(ij)}$ are only a "local patch" near (i,j) in the image x, and each $h_{(ij)}$ is computed in exactly the same way, but using patches in different locations on x. Now, a hidden *layer* consists of a collection of C filters; for each filter $c \in [1, C]$, we have a two-dimensional collection of hidden activations, $\{h_{c,(ij)}\}$, called a *channel*.

Example 6-2 : LeNet



input channels, etc?

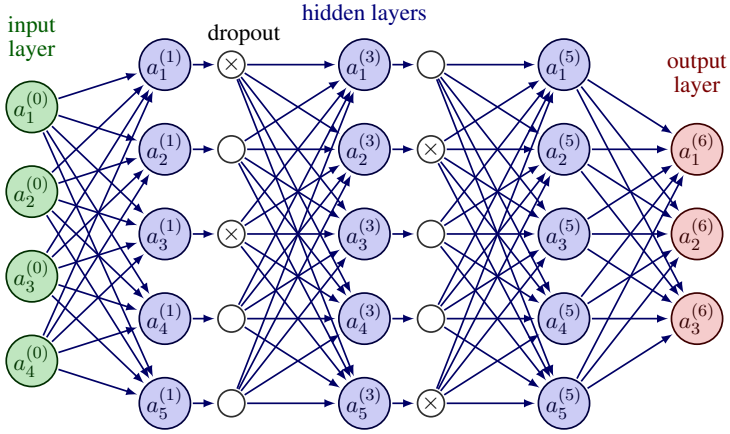
U-Networks?

Recurrent Networks?

Attention?

6.3 Other concepts

Dropout



Batch Normalization