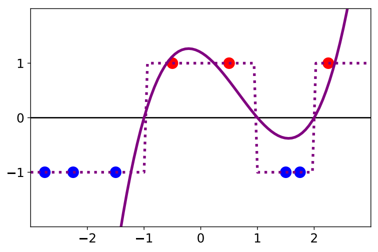
CHAPTER

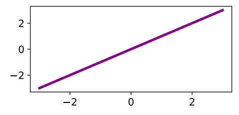
6 Neural Networks

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.

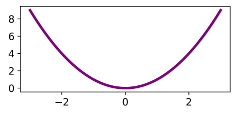
Example 6-1 : Non-linearly separable data

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:

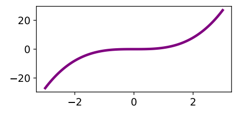




$\phi_1(x) = x$



$\phi_2(x) = x^2$

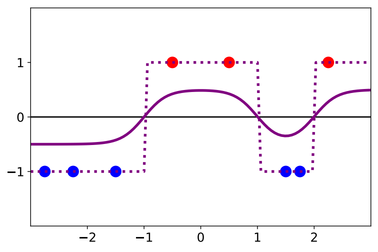


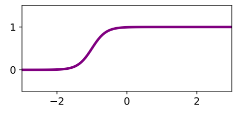
$\phi_3(x) = x^3$

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

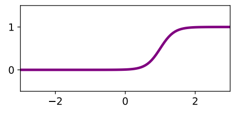
Although no linear function of the scalar feature x can correctly separate the training data, a cubic function of x (or equivalently, a linear function of degree-3 polynomial features of x) can.

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:

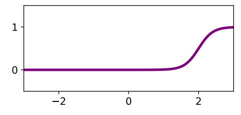




$\phi_1(x) = \sigma(5x + 5)$



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



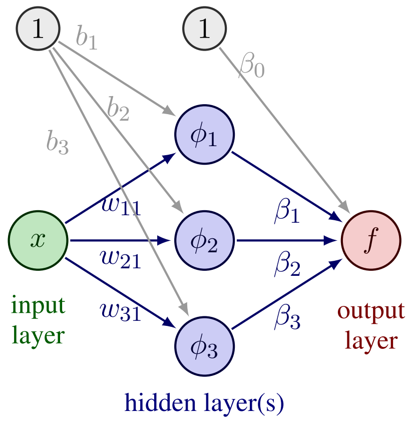
$\phi_3(x) = \sigma(5x - 10)$

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

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).

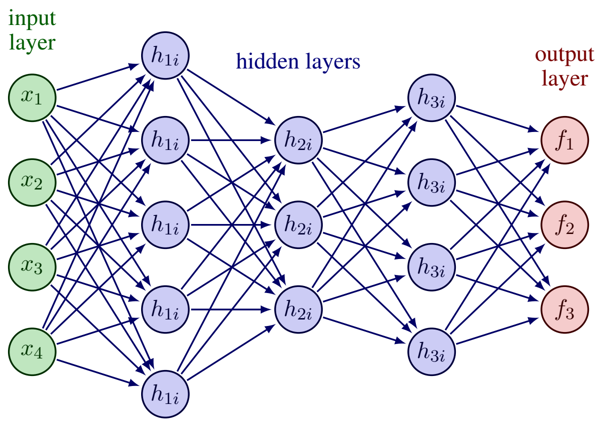
In defining our features ϕ_i and our output f, we have used the logistic sigmoid function $\sigma(r)$, to emphasize the similarity of these steps to a standard linear classifier, in which we used the logistic transform to define a smooth surrogate loss. Intuitively, when the linear response $r_i = b_i + w_{i1}x$ 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

The network architecture (number of layers, and sizes of each layer) determines the number of parameters of our model, and the set of functions that it is able to represent. Viewing the hidden layer outputs as (parameterized) features of the input suggests that, as with (fixed) feature expansions such as polynomials, wider neural networks will be more flexible and able to represent complex functions. The effect of depth is a bit more difficult to quantify, but tends to make the features themselves more flexible – for a single hidden layer, the features are essentially just (smooth) step functions, but in the same way that an output layer based on these features can produce a complex function, each node in a second hidden layer can produce a feature of similar complexity.

The number of parameters of a neural network also grows with its depth and width.

Example 6-2 : Number of parameters in a neural network

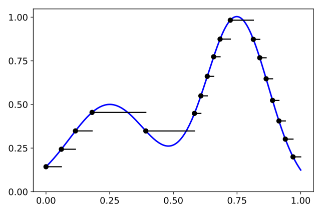
Let us again suppose that we have $L-1$ layers, with sizes $\{H_1, \ldots, H_{L-1}\}$, along with $H_0 = n$ input features and $H_L = K$ outputs. Then, we have $(n+1)H_1$ parameters in the first layer (nH_1 weights and H_1 bias terms), $(H_1+1)H_2$ parameters in the second layer, and so on, giving $\sum_{l=1}^{L}(H_{l-1}+1)H_l$ parameters total.

More holistically, if all layers have $H_l \approx H$, we have about $O(LH^2)$ parameters, i.e., the number of parameters in the model grows linearly with its depth, and quadratically with its width.

A classic motivational result of neural networks is their ability to be a *universal approximator*, i.e., that given sufficiently many hidden nodes, even a two-layer neural network can approximate any smooth function $f^*(x)$ up to an arbitrary tolerance ϵ. This is relatively trivial to see for scalar inputs x:

Example 6-3 : Neural approximation

Suppose that we wish to approximate $f^*(x)$ up to tolerance ϵ on some finite domain, say $x \in [0,1]$. Let $\mu^{(0)} = 0$, and $\mu^{(i+1)}$ be such that $f^*(x) \in [f^*(\mu^{(i)} - \epsilon, f^*(\mu^{(i)} + \epsilon]$ for all $x \in [\mu^{(i)}, \mu^{(i+1)}]$; in this way, there is some finite number m of points $\{\mu^{(i)}\}$ that discretize the domain of x into regions with nearly constant value $f^*(x)$. (An example discretization is illustrated at right.)



Thus, we can closely approximate f^* with a piecewise constant function $f(x)$. We can construct $f(x)$ using a two layer neural network as follows. We define each hidden node $h_j(x) = \sigma(\,-R\,(x - \mu^{(j)})\,)$, where R is a large constant, so that $h_j \approx \mathbb{1}[x \leq \mu^{(j)}]$. Finally, we define $\beta_0 = f^*(\mu^{(0)}) = f^*(0)$ and $\beta_j = f^*(\mu^{(j)}) - f^*(\mu^{(j-1)})$, so that

$$f(x) = \beta_0 + \sum_j \beta_j h_j = f^*(0) + (f^*(\delta) - f^*(0))h_1(x) + (f^*(2\delta) - f^*(\delta))h_2(x) + \ldots,$$

a telescoping series that evaluates to $f^*(x^{(j)})$ where $x^{(j)} \leq x \leq x^{(j+1)}$, so that $|f(x) - f^*(x)| \leq \epsilon$.

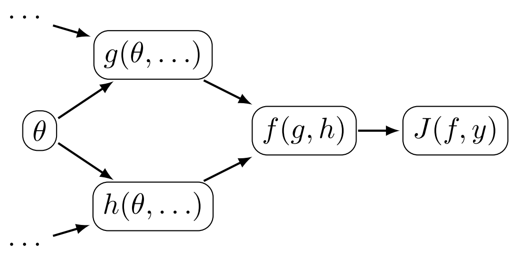
Obviously, this constructive illustration is not a particularly efficient representation of f^*, but it does assure us that as we increase the size of our neural network, we at least have the potential of being able to learn an optimal predictor. Next, we turn to learning the model from data: we must identify a good instance within this class, by finding parameter settings that match our examples in the training data.

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.



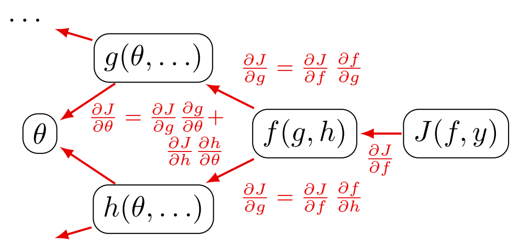
Forward pass

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**.

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,

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

In other words, the overall derivative of J is computed in terms of the values $(g_0, h_0, f_0, \ldots)$ 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 this 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 compute 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.


$\ldots$
$g(\theta, \ldots)$
$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial g}$
θ
$\frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial g}\frac{\partial g}{\partial \theta} + \frac{\partial J}{\partial h}\frac{\partial h}{\partial \theta}$
$f(g,h)$
$J(f,y)$
$\frac{\partial J}{\partial f}$
$h(\theta, \ldots)$
$\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f}\frac{\partial f}{\partial h}$
$\ldots$
Backward pass

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:

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

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:

$$\begin{aligned} \frac{\partial J}{\partial \beta_{jk}} &= \sum_{k'} \Big(\frac{\partial J}{\partial f_{k'}}\Big)\Big(\frac{\partial f_{k'}}{\partial s_{k'}}\Big)\Big(\frac{\partial s_{k'}}{\partial \beta_{jk}}\Big) \\ &= \Big((-2)(y_{k'} - f_{k'})\Big)\Big(a_2'(s_k)\Big)\Big(h_j\Big) \end{aligned} \tag{6.2}$$

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,

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

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

$$\frac{\partial s_{k'}}{\partial w_{ij}} = \sum_{j'} \left(\frac{\partial s_{k'}}{\partial h_{j'}}\right) \left(\frac{\partial h_{j'}}{\partial r_{j'}}\right) \left(\frac{\partial r_{j'}}{\partial w_{ij}}\right)$$
$$= \Big(\beta_j\Big) \Big(a_1'(r_j)\Big) \Big(x_i\Big)$$

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

$$\frac{\partial J}{\partial w_{ij}} = \sum_{k'} \Big((-2)(y_{k'} - f_{k'})\Big) \Big(a_2'(s_k)\Big) \Big(\beta_j\Big) \Big(a_1'(r_j)\Big) \Big(x_i\Big). \tag{6.3}$$

Again, for the bias parameters, 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.)

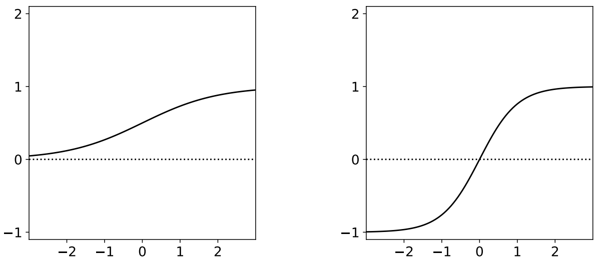
Thus, each node is performing a very simple operation: having saved its input values during the forward pass, it computes the derivative of its output with respect to each input, and passes this back to its respective input, multiplied by the sum of derivatives recieved from its outputs. After a backward 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:

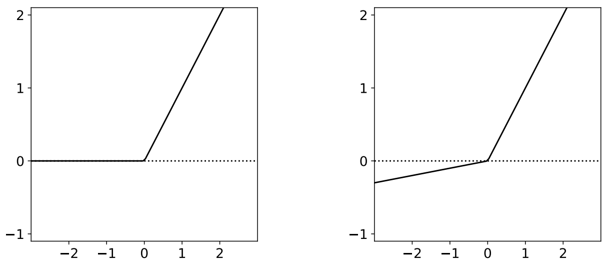


$a(r) = \frac{1}{1+\exp(-r)}$ Logistic sigmoid

$a(r) = \frac{1-\exp(-2r)}{1+\exp(-2r)}$ 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) $\quad$ $a(r) = \max[r, \epsilon\, r]$ 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, $\underline{r}_L = [r_{L1}, \ldots, r_{LC}]$ and outputs a vector of values $\underline{f}$:

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

C classes or K classes? 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),

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

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

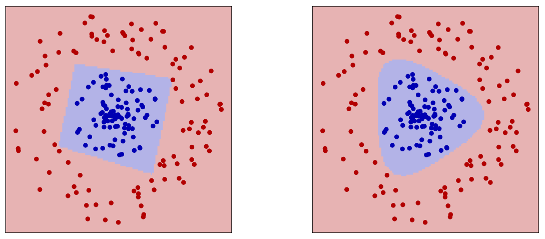
Practical considerations. Historically, saturating activation functions such as the logistic or hyperbolic tangent functions were most common in neural networks. However, with modern, large networks there are practical reasons to use ReLU or leaky ReLU activation functions. Evaluating the activation 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 6-4 : Activation functions

The type of activation functions we select can strongly influence the form that our function f can take on. For example with ReLU activations, the outputs of the first hidden 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.

To illustrate this point, consider a small two-dimensional data set (shown at right). A simple two-layer neural network with three hidden nodes and either ReLU or logistic activation functions on the hidden nodes is able to separate the data; but the two choices produce different types of decision functions: piecewise linear boundaries for ReLU, compared to smoother boundaries for the model using logistic activations.


ReLU Logistic

6.2 Special Architectures

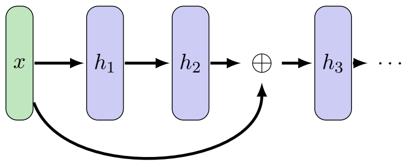
Residual Networks

Residual network blocks are a common neural network sub-structure in deep models. The basic idea of a residual block is to use several hidden layers to represent a *perturbation* to the input, rather than a fundamental transformation (or replacement) of the input. So, instead of our usual hidden layer transformations, we might define a block of two hidden layers to transform x by:

$$h(x) = x + h_2(\,h_1(x\,;\,W_1)\,;\,W_2) \qquad \text{vs.} \qquad h(x) = h_2(\,h_1(x\,;\,W_1)\,;\,W_2)$$

(Residual 2-layer block) (Standard 2-layer block)

We can diagram this structure by placing a **skip connection** from x to a summation point (indicated by $\oplus$) at the end of the block, which then is used as an input to subsequent layers of the network.



There are several reasons why this residual structure might be better for training. From one perspective, suppose that we place a standard regularization (say, L_2) on the weights W_l. This regularization will encourage our weights to be small, and shifts the resulting features $h_2(h_1(x))$ towards the all-zero vector. In contrast, the residual network shifts the resulting features transform toward the identity (using the original features x). If these features are used, for example as inputs to the final (output) layer of the network, strong regularization on the W_l will make our model more like a linear classifier – likely to be a more practical notion of a "simple classifier" than one that zeroes out the input features before classifying.

The second reason why skip layers are helpful is to combat the so-called *vanishing gradient* phenomenon. We saw in our backpropagation calculation (6.2)–(6.3) that, as we moved back from one layer to the next, our gradients involved multiplying by the activation slopes (e.g., $a_2'(s)$ and $a_1'(r)$), and also summing over the subsequent layer's hidden nodes (e.g., the summation over k' in $\partial J/\partial w_{ij}$). Both operations have a tendency to decrease the magnitude of the gradient vectors: the slopes a' are typically smaller than one (for example, for the logistic σ, they are $\leq \frac{1}{4}$), and if the gradient directions of a given layer's hidden nodes are random with zero mean (in average across the hidden nodes), the preceding layer's gradients will tend to be closer to zero, thanks to the sum over the nodes. The vanishing gradient effect makes it increasingly difficult to train the earliest layers of the network, since they are so distantly related to the actual output f.

Residual blocks and their associated skip connections sidestep this issue. Since the output of the block is directly related to its input, x, the gradients passed backward to earlier layers will stay large, and not be decreased by the activation functions' slopes or the averaging effect of the weights.

As described so far, the residual "skip" structure assumes that the output of h_2 is the same size as the residual block's input, x. If this is not the case, we can extend our definition of the skip layer to include a minor transformation on x. If the dimension of h_2 is larger than x, we can simply extend x by zeros; if its dimension is smaller, we can include skip weights W_s to project x down to the desired dimension before re-adding, e.g.,

$$h(x) = W_s \odot x + h_2(\, h_1(x\,;\, W_1)\,;\, W_2).$$

Convolutional Neural Networks

Convolutional neural network blocks have been very successful on problems whose inputs are structured, for example images (a two-dimensional organization of pixel intensities) or time series (a one-dimensional organization of sequential observations). The idea is to exploit the locality of useful patterns in the input by defining our hidden nodes in terms of the responses of contiguous, local subsets of the input.

As an explicit example we consider the image processing task of recognizing handwritten digits. Our input x is a 28×28 image of intensity values, whose individual dimensions we can index as $x_{(i,j)}$, where (i, j) is the pixel's position in the image and $x_{(i,j)}$ is that pixel's intensity. Now, we can define a hidden node $h_{(i,j)}$ that depends only on a small patch of x around location (i, j). We parameterize the hidden nodes by a collection of weights $w_{k,l}$, and define the hidden response as,

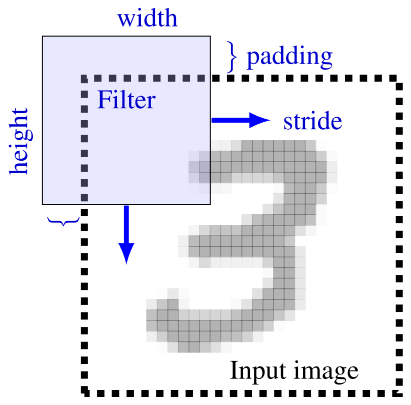
$$h_{(i,j)} = a\Big(\sum_{k\in[-K,K]} \sum_{l\in[-L,L]} w_{k,l}\, x_{(i+k,j+l)} \Big). \tag{6.4}$$

We can see that the $(2K + 1) \cdot (2L + 1)$ weights multiply an equally sized patch of the pixels surrounding (i, j); these weights are often called a **filter**, due to their form's similarity to operations in signal processing.

Now, we apply (6.4) to compute a hidden response $h_{(i,j)}$ for as many possible center locations (i, j) as we would like. Notice that these responses are all defined by the same weights, but different subsets of the features x; so even though there could be a large number of hidden nodes $h_{(i,j)}$ (approximately as many as the size of the input image), the number of parameters (weights) remains small, $(2K + 1) \cdot (2L + 1)$. Moreover, these responses $h_{(i,j)}$ also have a spatial organization based on (i, j); we can assemble them into an "image" of activation values, which we call a **channel**. Finally, a hidden **layer** consists of a collection of C channels; for each channel c, we have a collection of $(2K + 1) \cdot (2L + 1)$ weights, $w_{c,k,l}$, and a two-dimensional collection of hidden activations, $h_{c,(i,j)}$. Finally, our input could also have multiple channels – for example, $x_{(i,j)}$ could have three values to express an RGB color, or the input could be the output of a preceding hidden layer with multiple channels; in this case, we typically sum over the channels as well, so that:

$$h_{c,(i,j)} = a\Big(\sum_{k\in[-K,K]} \sum_{l\in[-L,L]} \sum_{c'} w_{c,c',k,l}\, x_{c',(i+k,j+l)} \Big). \tag{6.5}$$

In general, we will typically express the structure of a convolutional layer in terms of a few standard quantities: the number of output channels (or filters); the size of each output channel's filter (width and height); the "stride", or how much we should shift the filter's location (i, j) from one hidden node to the next; and finally "padding" at the boundary of the input, which extends the input slightly (say, by padding with zeros) to allow the filter's center to be applied even near the edge of the input image. The number of parameters (weights) in the layer is the product of the filter width and height, and number of input and output channels, while the input width and height, the padding and stride affect the number of hidden nodes per channel, by increasing or reducing the number of locations where each filter is applied to the input.



The filters can be intuitively understood to be local feature detectors. Once trained, the filter will result in a high activation value when its high-magnitude weights are aligned with high-magnitude pixel values of the same sign in the input image.

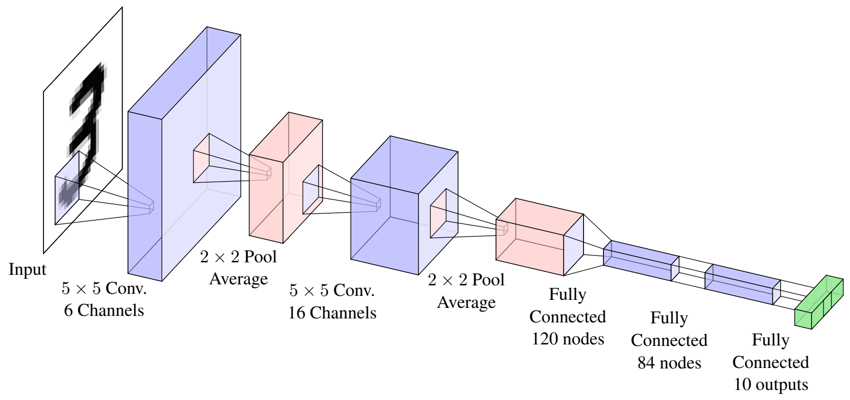
Example

Pooling layers. Convolutional layers produce a large number of activation values, similar to the size of the input image. However, we are usually trying to aggregate information about the overall image to eventually make a low-dimensional prediction, such as the class value (e.g., which of the 10 possible digits is represented in the input). Pooling layers combine convolutional responses across a local region to reduce the size (height, width) of the input layer. Common pooling layers combine the responses by either **average pooling** or **max pooling**. Average pooling simply averages the responses over a $K \times L$ window, while max pooling selects the highest activation value within the window. Average pooling can be interpreted as a special type of convolutional layer (with no additional parameters to learn), while max pooling can be interpreted as a feature selection step, keeping only the filter responses from the previous layer that are most active in each locale. By selecting a stride greater than one, the resulting image of responses becomes smaller. For example, we might pool over 2×2 windows with a stride of 2, halving the width and height of the channels.

non-max suppression intuition?

Example 6-5 : LeNet

An early and now-classic example of a convolutional neural network was given in **?**, applied to the MNIST dataset of handwritten digits. It consists of a 5×5 convolutional layer with 6 channels, applied to the 28×28 input image with 2 pixels of padding, resulting in $6 \times 28 \times 28$ hidden activation values. These are then pooled by averaging over 2×2 windows to become $6 \times 14 \times 14$. The model follows with another 5×5 convolutional layer with 16 channels (this time with no padding), and 2×2 average pooling, with output $16 \times 5 \times 5$. From this point, the model becomes fully connected: all $16 \cdot 5 \cdot 5 = 400$ hidden nodes are connected to 120 hidden nodes at the next layer, followed by 84 hidden nodes, and finally 10 output nodes. We illustrate the resulting model with a diagram:



note: fully connected layers where layer shape = Cx1x1 (sum over channels = all hidden nodes), or where filter shape = layer shape, with no padding.

input channels, etc?

Attention

Another now-common architecture, particularly in language models, is that of **attention**.

- Input a collection of tokens x_i, but may be vector-valued (e.g., color channels in an image, or vector representation of a word)
- Compute two quantities from each x_i, a **key** and **value**: $k_i = K \odot x_i$ and $v_i = V \odot x_i$.
- Now, given a **query** vector q_j, we compute the output vector h_j as:

  $$h_j = \text{softmax}([k_i \odot q_j]) \odot [v_i],$$

  i.e., we compute the list of all key-query responses $k_i \odot q_j$ for each input i, apply a softmax function (exponentiating the elements and then normalizing the vector), and use these values to compute the weighted average over the list of value vectors. The output h_j will be vectors of the same size as the v_i.
- Where does the query come from? In **self-attention** layers, we define a query vector for each x_i: $q_i = Q \odot x_i$. Then, every input x_j has an associated output h_j, so that the attention can be viewed as a transformation of each x_j in a manner that depends on all the other x_i.
- We can generalize this set-up by defining a query, key, and value *network* (rather than a simple matrix), typically a few layers of standard (fully-connected) components.
- Finally, we call the collection (K, V, Q) a **head**, and may use multiple heads in a single layer, analogously to using multiple filters to define several output channels in a convolutional layer.

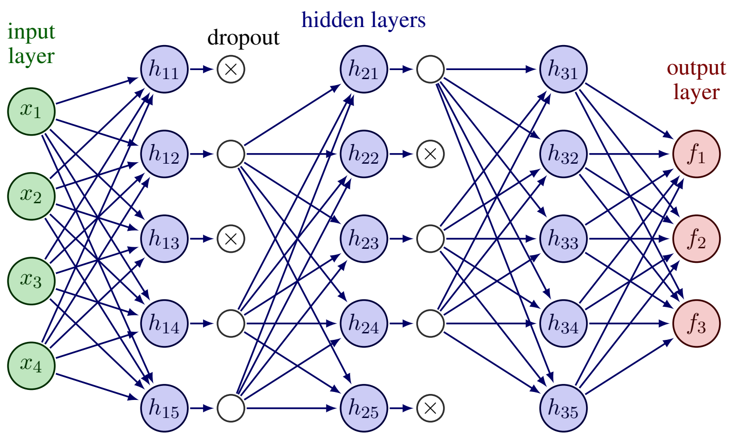
diagram

6.3 Other concepts

Dropout

A **dropout** layer stochastically censors (zeros the output of) a small fraction p of the preceding layer's nodes. Typically, we perform this censoring step randomly each time a data point is fed forward through the network to find its output $f(x)$ and corresponding loss J, so that each data point experiences a slightly different network. We can view this as encouraging a form of redundancy in the hidden nodes' representations: if a particular hidden node is capturing something important about the input, that information should also appear in other hidden nodes, since otherwise it is easily lost by the censoring. In this sense, we can view dropout as a mechanism to help avoid overfitting.

After training, we typically do not want our predictions to continue to be stochastic, so we turn off the random censoring process. However, as presented this leaves us with a minor issue – during training, the average input value to the next layer is smaller than it will be during prediction. To see this, imagine that our dropout probability was $p = \frac{1}{2}$. Then, in a typical training step, each hidden node h_j recieves a sum of $(1-p)\,n = n/2$ inputs (the rest having been zeroed out); but at predict time, it will recieve a sum over all n inputs, which on average will be twice as large. To compensate, we scale the outputs of the dropout layer during training by a factor of $1/(1-p)$, to compensate for some outputs being zeroed. Then, at prediction time, we can simply set $p = 0$.



Batch Normalization

add

We saw "standard" scaling to zero mean and unit variance; often helps with training. Although we can do this at the input, x, we have no guarantees that the intermediate values at some inner layer l, h_l, are also scaled "nicely". Batch norm inserts a normalization operation in the middle of the network, subtracting a value μ_l and scaling by factor of σ_l.

Imagine that we passed a large amount of training data through the network; then, we could estimate the mean μ_l and standard deviation σ_l of the activations h_l, and use these values to scale the outputs, $\tilde{h}_l = (h_l - \mu_l)/\sigma_l$, to have zero mean and unit variance. However, as soon as we begin training the weights of the network, our scaling parameters will change.

The idea of batch normalization is to use the mini-batch of data during stochastic gradient to re-estimate μ_l and σ_l "on the fly" to make each mini-batch normalized.

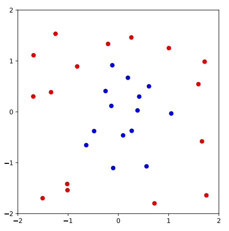
At predict time, we cannot continue to estimate μ and σ – imagine if we were given only one prediction point; then it would always be zero! Instead, we "lock" the values of μ and σ to their most recent values. A slightly more principled method is to partially update μ and σ during training (for example, taking a step toward the empirical mean and variance of each mini-batch); following an appropriate update schedule can allow the values of mu and σ to converge during training along with the model weights.

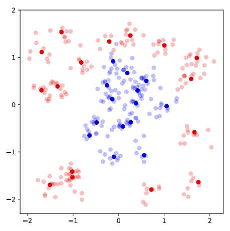
perspective as regularization; coupling/correlating of data in the mini-batch

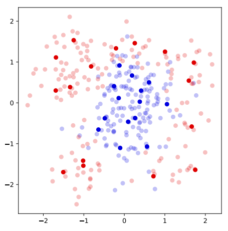
Example 6-6 : Batch Normalization as Regularization

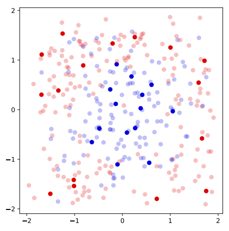
fill in

We can visualize the distribution of inputs that our network would see from a batch norm operation performed on various mini-batch sizes, B, for a data set of size $m = 30$ that has been pre-normalized for visualization. If we sample $B = 20$ points from the full data set (a) and then normalize them, we see the a distribution of inputs illustrated in panel (b), with the original data also included for perspective. Each input mini-batch contains most but not all the data points; normalizing this batch then moves the points slightly, to center and scale the subset of data. As the mini-batch size decreases to $B = 10$ in panel (c), and $B = 5$ in panel (d), the perturbations of this normalization step become larger, with the practical effect of "blurring" the data positions by larger amounts, and thus encouraging smoother outputs from our learner.


(a) Data


(b) $B = 20$


(c) $B = 10$


(d) $B = 5$