Deep Learning
Ch6.2.2.3 CH6.2.2.4

Authors: Ian Goodfellow, Yoshua Bengio, Aaron Courville

Reporter: Bo-Zen Xiao

2017-04-05

Contents

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.
- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.
- In the case of binary variables, we wished to produce a single number

$$\hat{y} = P(y = 1|x).$$

- Because this number needed to lie between 0 and 1, and because we wanted the logarithm of the number to be well-behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number $z = log\widetilde{P}(y = 1|x)$. Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\widehat{y}$, with $\widehat{y}_i = P(y = i|x)$. We require not only that each element of $\widehat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$z = W^\top h + b,$$

- where $z_i = log\widetilde{P}(y = i|x)$. The softmax function can then exponentiate and normalize z to obtain the desired $\widehat{y}$. Formally, the softmax function is given by

$$softmax(z)_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- As with the logistic sigmoid, the use of the exp function works very well when training the softmax to output a target value y using maximum log-likelihood. In this case, we wish to maximize $logP(y = i; z) = logsoftmax(z)_i$. Defining the softmax in terms of exp is natural because the log in the log-likelihood can undo the exp of the softmax:

$$logsoftmax(z)_i = z_i - log \sum_j exp(z_j)$$

- The first term of Eq. shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of Eq. becomes very small. When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of z to be pushed down.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- To gain some intuition for the second term, $log \sum_j exp(zj)$, observe that this term can be roughly approximated by $max_j z_j$. This approximation is based on the idea that $exp(z_k)$ is insignificant for any z_k that is noticeably less than $max_j z_j$.
- The intuition we can gain from this approximation is that the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $log \sum_j exp(z_j) \approx max_j z_j = z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- So far we have discussed only a single example. Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict the fraction of counts of each outcome observed in the training set:

$$softmax(z(x;\theta))_i \approx \frac{\sum_{j=1}^{m} 1_{y^{(j)}=i, x^{(j)}=x}}{\sum_{j=1}^{m} 1_{x^{(j)}=x}}$$

- Because maximum likelihood is a consistent estimator, this is guaranteed to happen so long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units, and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions. To understand why these other loss functions can fail, we need to examine the softmax function itself.
- Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. In the case of the softmax, there are multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- To see that the softmax function responds to the difference between its inputs, observe that the softmax output is invariant to adding the same scalar to all of its inputs:

$$softmax(z) = softmax(z + c)$$

- Using this property, we can derive a numerically stable variant of the softmax:

$$softmax(z) = softmax(z - \max_i z_i)$$

- The reformulated version allows us to evaluate softmax with only small numerical errors even when z contains extremely large or extremely negative numbers. Examining the numerically stable variant, we see that the softmax function is driven by the amount that its arguments deviate from $max_i z_i$.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- An output $softmax(z)_i$ saturates to 1 when the corresponding input is maximal $(z_i = max_i z_i)$ and z_i is much greater than all of the other inputs. The output $softmax(z)_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater. This is a generalization of the way that sigmoid units saturate, and can cause similar difficulties for learning if the loss function is not designed to compensate for it.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- The argument z to the softmax function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of z, as described above using the linear layer $z = W^\top h + b$. While straightforward, this approach actually overparametrizes the distribution. The constraint that the n outputs must sum to 1 means that only $n - 1$ parameters are necessary; the probability of the n-th value may be obtained by subtracting the first $n - 1$ probabilities from 1.
- We can thus impose a requirement that one element of z be fixed. For example, we can require that $z_n = 0$. Indeed, this is exactly what the sigmoid unit does. Defining $P(y = 1|x) = \sigma(z)$ is equivalent to defining $P(y = 1|x) = softmax(z)_1$ with a two-dimensional z and $z_1 = 0$. Both the $n - 1$ argument and the n argument approaches to the softmax can describe the same set of probability distributions, but have different learning dynamics. In practice, there is rarely much difference between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized version.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of *winner-take-all* (one of the outputs is nearly 1 and the others are nearly 0).

6.2.2.3 Softmax Units for Multinoulli Output Distributions

- The name “softmax” can be somewhat confusing. The function is more closely related to the argmax function than the max function. The term “soft” derives from the fact that the softmax function is continuous and differentiable. The argmax function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a “softened” version of the argmax. The corresponding soft version of the maximum function is $softmax(z)^\top z$. It would perhaps be better to call the softmax function “softargmax”, but the current name is an entrenched convention.

6.2.2.4 Other Output Types

- The linear, sigmoid, and softmax output units described above are the most common. Neural networks can generalize to almost any kind of output layer that we wish. The principle of maximum likelihood provides a guide for how to design a good cost function for nearly any kind of output layer.
- In general, if we define a conditional distribution $p(y|x;\theta)$, the principle of maximum likelihood suggests we use -log$p(y|x;\theta)$ as our cost function.
- In general, we can think of the neural network as representing a function $f(x;\theta)$. The outputs of this function are not direct predictions of the value y. Instead, $f(x;\theta) = \omega$ provides the parameters for a distribution over y. Our loss function can then be interpreted as -log$p(y;\omega(x))$.

6.2.2.4 Other Output Types

- For example, we may wish to learn the variance of a conditional Gaussian for y , given x. In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations y and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(y|x)$ that is controlled by $\omega = f(x;\theta)$. The negative log-likelihood -log$p(y;\omega(x))$ will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance.

6.2.2.4 Other Output Types

- In the simple case where the standard deviation does not depend on the input, we can make a new parameter in the network that is copied directly into ω. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma^2}$, depending on how we choose to parametrize the distribution. We may wish our model to predict a different amount of variance in y for different values of x. This is called a heteroscedastic model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(x;\theta)$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in Eq.

$$\mathcal{N}(x;\mu,\beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} exp(-\frac{1}{2}\beta(x-\mu)^2)$$

In the multivariate case it is most common to use a diagonal precision matrix

$$diag(\beta).$$

6.2.2.4 Other Output Types

- This formulation works well with gradient descent because the formula for the log-likelihood of the Gaussian distribution parametrized by β involves only multiplication by β_i and addition of log β_i . The gradient of multiplication, addition, and logarithm operations is well-behaved. By comparison, if we parametrized the output in terms of variance, we would need to use division. The division function becomes arbitrarily steep near zero. While large gradients can help learning, arbitrarily large gradients usually result in instability. If we parametrized the output in terms of standard deviation, the log-likelihood would still involve division, and would also involve squaring. The gradient through the squaring operation can vanish near zero, making it difficult to learn parameters that are squared.

6.2.2.4 Other Output Types

- Regardless of whether we use standard deviation, variance, or precision, we must ensure that the covariance matrix of the Gaussian is positive definite. Because the eigenvalues of the precision matrix are the reciprocals of the eigenvalues of the covariance matrix, this is equivalent to ensuring that the precision matrix is positive definite. If we use a diagonal matrix, or a scalar times the diagonal matrix, then the only condition we need to enforce on the output of the model is positivity. If we suppose that a is the raw activation of the model used to determine the diagonal precision, we can use the softplus function to obtain a positive precision vector: $\beta = \xi(a)$. This same strategy applies equally if using variance or standard deviation rather than precision or if using a scalar times identity rather than diagonal matrix.

6.2.2.4 Other Output Types

- It is rare to learn a covariance or precision matrix with richer structure than diagonal. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive-definiteness of the predicted covariance matrix. This can be achieved by writing $\sum(x) = \mathbf{B}(x)\mathbf{B}^\top(x)$, whereB is an unconstrained square matrix. One practical issue if the matrix is full rank is that computing the likelihood is expensive, with a $d \times d$ matrix requiring $O(d^3)$ computation for the determinant and inverse of $\sum(x)$ (or equivalently, and more commonly done, its eigendecomposition or that of $\mathbf{B}(x)$).

6.2.2.4 Other Output Types

- We often want to perform multimodal regression, that is, to predict real values that come from a conditional distribution $p(y|x)$ that can have several different peaks in y space for the same value of x. In this case, a Gaussian mixture is a natural representation for the output . Neural networks with Gaussian mixtures as their output are often called *mixture density networks*. A Gaussian mixture output with n components is defined by the conditional probability distribution

$$p(y|x) = \sum_{i=1}^{n} p(c = i|x)\mathcal{N}(y; \mu^{(i)}(x), \Sigma^{(i)}(x))$$

6.2.2.4 Other Output Types

- The neural network must have three outputs: a vector defining $p(c = i|x)$, a matrix providing $\mu^{(i)}(x)$ for all i, and a tensor providing $\Sigma^{(i)}(x)$ for all i. These outputs must satisfy different constraints:
  (1) Mixture components $p(c = i|x)$: these form a multinoulli distribution over the n different components associated with **latent variable c**, and can typically be obtained by a softmax over an n-dimensional vector, to guarantee that these outputs are positive and sum to 1.
  (*We consider c to be latent because we do not observe it in the data: given input x and target y , it is not possible to know with certainty which Gaussian component was responsible for y, but we can imagine that y was generated by picking one of them, and make that unobserved choice a random variable.*)

6.2.2.4 Other Output Types

(2) Means $\mu(i)(x)$: these indicate the center or mean associated with the i-th Gaussian component, and are unconstrained (typically with no nonlinearity at all for these output units). If y is a d-vector, then the network must output an $n \times d$ matrix containing all n of these d-dimensional vectors. Learning these means with maximum likelihood is slightly more complicated than learning the means of a distribution with only one output mode. We only want to update the mean for the component that actually produced the observation. In practice, we do not know which component produced each observation. The expression for the negative log-likelihood naturally weights each example's contribution to the loss for each component by the probability that the component produced the example.

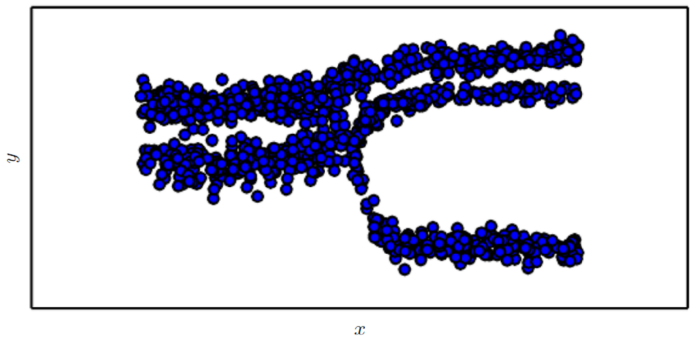
6.2.2.4 Other Output Types

(3) Covariances $\Sigma^{(i)}(x)$: these specify the covariance matrix for each component i. As when learning a single Gaussian component, we typically use a diagonal matrix to avoid needing to compute determinants. As with learning the means of the mixture, maximum likelihood is complicated by needing to assign partial responsibility for each point to each mixture component. Gradient descent will automatically follow the correct process if given the correct specification of the negative log-likelihood under the mixture model.

6.2.2.4 Other Output Types

- It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be unreliable, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to *clip gradients* (see Sec. 10.11.1) while another is to scale the gradients heuristically (Murray and Larochelle, 2014).
- Gaussian mixture outputs are particularly effective in generative models of speech (Schuster, 1999) or movements of physical objects (Graves, 2013). The mixture density strategy gives a way for the network to represent multiple output modes and to control the variance of its output, which is crucial for obtaining a high degree of quality in these real-valued domains. An example of a mixture density network is shown in Fig. 6.4.

6.2.2.4 Other Output Types



- Figure 6.4: Samples drawn from a neural network with a mixture density output layer. The input x is sampled from a uniform distribution and the output y is sampled from $p_{model}(y|x)$. The neural network is able to learn nonlinear mappings from the input to the parameters of the output distribution. These parameters include the probabilities governing which of three mixture components will generate the output as well as the parameters for each mixture component. Each mixture component is Gaussian with predicted mean and variance. All of these aspects of the output distribution are able to vary with respect to the input x, and to do so in nonlinear ways

6.2.2.4 Other Output Types

- In general, we may wish to continue to model larger vectors y containing more variables, and to impose richer and richer structures on these output variables. For example, we may wish for our neural network to output a sequence of characters that forms a sentence. In these cases, we may continue to use the principle of maximum likelihood applied to our model $p(y; \omega(x))$, but the model we use to describe y becomes complex enough to be beyond the scope of this chapter. Chapter 10 describes how to use recurrent neural networks to define such models over sequences, and Part III describes advanced techniques for modeling arbitrary probability distributions.

Thank You for your Attention