medium.com

A guide to generating probability distributions with neural networks

Sam Blake

11–14 minutes

A few months ago we published an article that introduced the concept of confidence intervals and showed how, by sampling "thinned" networks using dropout, we go about generating them for pre-trained neural network regression models.

What if instead, as suggested by some of the readers, we wanted to train a model that can predict its own (un)certainty? In this article we will show you how we do just that, using Tensorflow with the Keras functional API to train a neural network that predicts a probability distribution for the target variable.

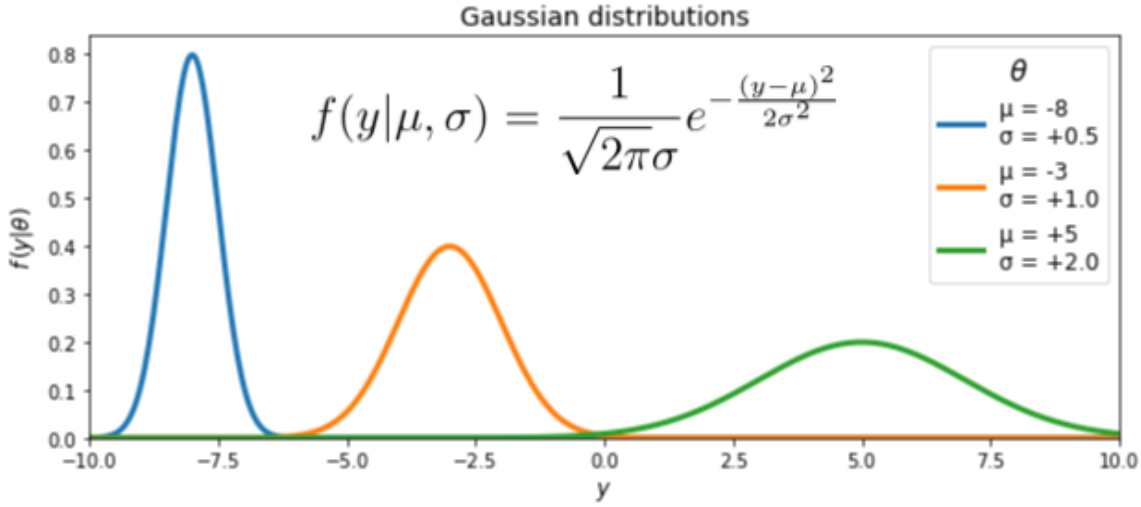
We'll be using the exact same dataset as the first article — traffic flow — and modelling the target variable with a negative binomial distribution. However, I'm going to try and explain each step in a general manner, so that hopefully you can extend the process to any dataset and distribution of your choosing.

Choosing a probability distribution

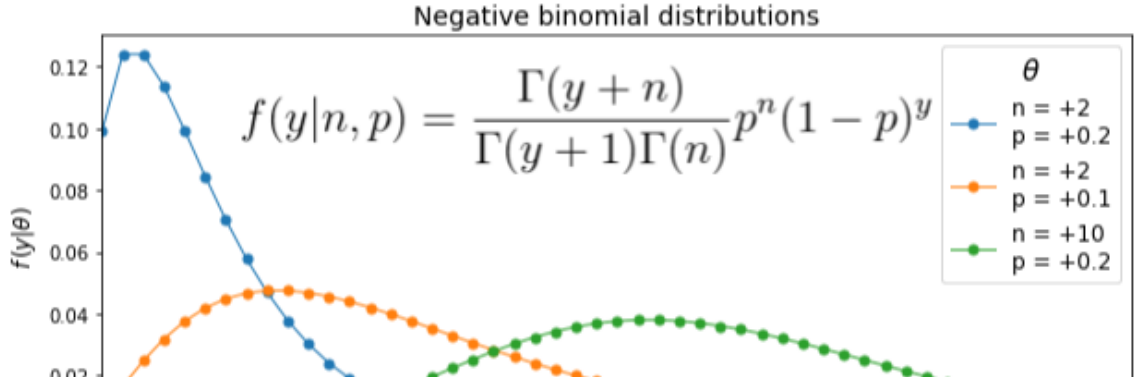
To train a model that predicts a probability distribution, we must first decide on what general shape the distribution will take. There are a lot of possible distributions to choose from, but that choice will be in part determined by any constraints on the values the target variable is allowed to take. Is the target discrete or continuous? Does it have fixed upper and lower limits, or an infinite range?

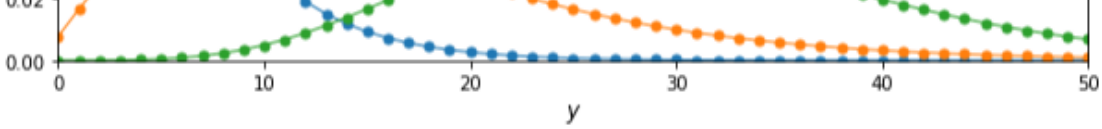
Let's take a few examples from the kind of projects we often work on at HAL24K.

Water levels are continuous and effectively unbounded (within the region of probable values). For these we use a **Gaussian** distribution.

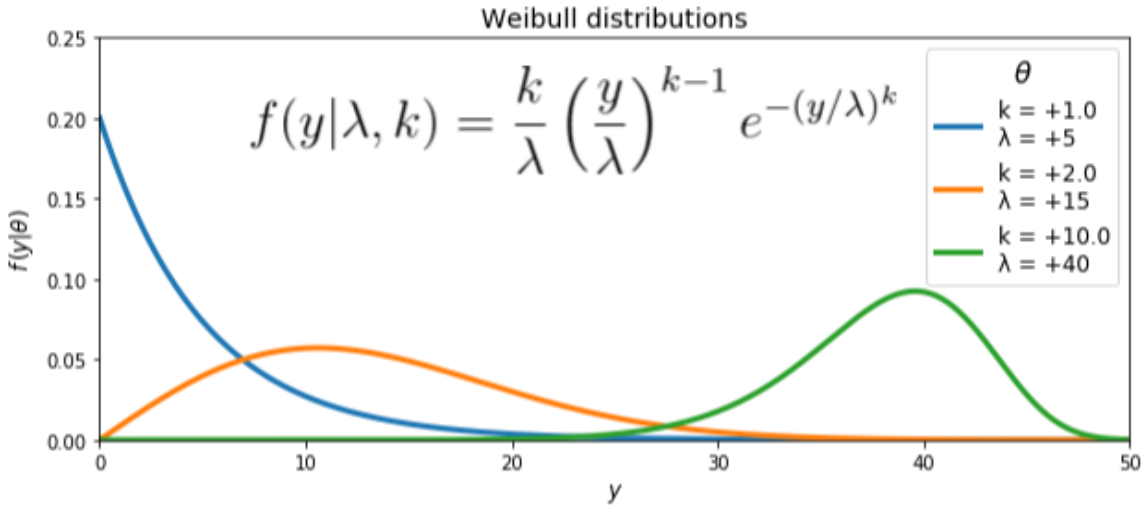


For **demand forecasting** the possible values are discrete (only integer values are allowed) and have a lower bound of zero. For these we use a **negative binomial** distribution.





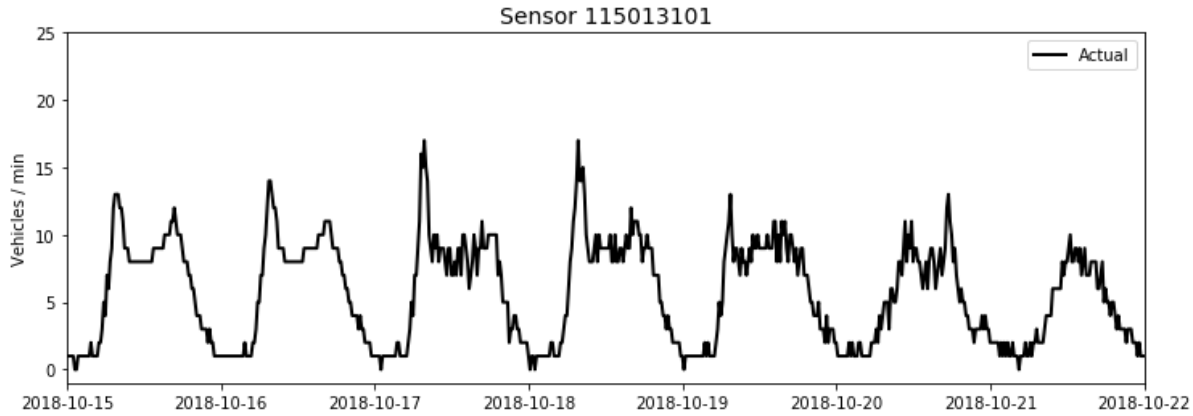
Time to failure is continuous and positive. For these we use a Weibull distribution.



Whatever type of probability distribution we decide upon, it can be expressed as a function *f(y|θ)*, as in the plots above. Here *y* represents all possible values of the target variable — we will use *Y* to designate the actual value — and *θ* represents the parameter(s) which set the exact shape of the distribution.

With our usual regression models we would take a set of inputs, *X*, and train our model to predict the actual value of the target variable, *Y*. To predict instead the distribution of possible values for *Y*, *f(y|θ)*, we will train our model to predict the value(s) of *θ* for our chosen distribution.

Let's take a look at our example dataset of traffic flows.



We can see these are discrete and positive (there's no such thing as minus half a car) and so we'll model them with a negative binomial distribution. We gave the formula for this above, but for the sake of clarity let's look at it, and the constraints on its values, again.

$$f(y|n,p) = \frac{\Gamma(y+n)}{\Gamma(y+1)\Gamma(n)} p^n (1-p)^y$$

$$\{y \in \mathbb{N}\};\ \{n \in \mathbb{R}, 0 < n\};\ \{p \in \mathbb{R}, 0 < p < 1\}$$

The negative binomial distribution is described by two parameters, *n* and *p*. These are what we will train our network to predict. The first of these, *n*, must be positive, while the second, *p*, must fall between 0 and 1. *y* itself can only take integer values, and so *f(y|n,p)* gives us directly the probability for each possible value of the target variable: *f(y|n,p) = P(Y=y)* (i.e. the probability mass function).

Building the network

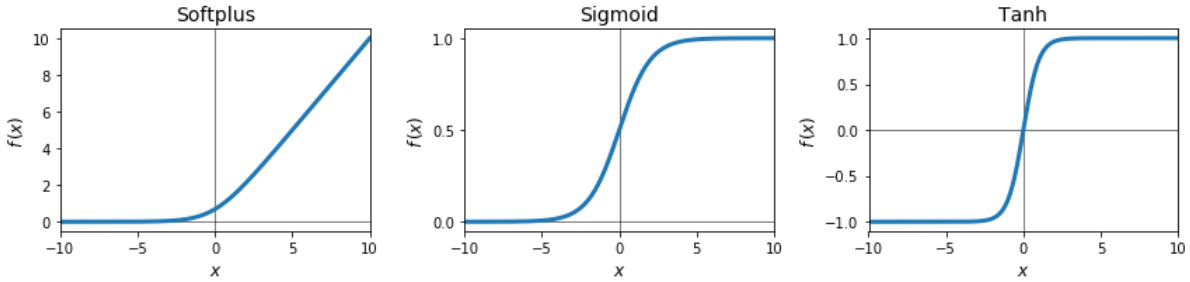
Now that we have decided on what it is we want our model to predict, we can go about constructing our neural network. If we were performing a regression of the target variable, our model — constructed using the functional

API of Keras — would look something like the following:

Here we define our initial inputs and some kind of model architecture, then add a final fully connected layer at the end whose single output will be our target variable, *Y*. The exact form of the inputs and architecture will vary from model to model (hence the generic `Layer1`, `Layer2` etc above) and lies beyond the scope of this article. (For our traffic flow example we're using an LSTM sequence to sequence model, as before). To adapt the above model to return a probability distribution, all we have to do is change that final fully connected layer.

To perform this adaptation — from predicting the target variable *Y* to predicting the probability distribution $f(y|\theta)$ via the parameters θ — there are two changes we need to make. Firstly, we need to increase the number of outputs to match the number of parameters of θ. We can easily do this by increasing the number of units when we define the Dense layer.

Secondly we need to apply any necessary limits to these parameters, as the outputs of the Dense layer are theoretically unbounded. We can do this by applying activation functions after the Dense layer. A few useful examples are shown below: a softplus activation will restrict a parameter to positive values only; a sigmoid activation will restrict a parameter to between 0 and 1; a tanh activation will restrict a parameter to between -1 and 1.



Often we will need to apply different activation functions to the different parameters of θ and so we cannot use the `activation` parameter of the Dense layer itself. Instead we will define a custom layer in our network that applies the correct activation to each parameter. We can do this by writing a custom function and wrapping it in a Keras Lambda layer.

Let's look at a concrete example for our traffic flow predictions. For a negative binomial distribution we need to return the two parameters *n* and *p*, and so our final Dense layer has 2 units. *n* must be positive, so we use a softplus activation. *p* must be between 0 and 1, so we use a sigmoid activation. Our negative binomial function then looks like so:

To construct our negative binomial model, we simply take the previous regression model, increase the number of units in the final Dense layer to 2, and add on the negative binomial layer (i.e. the above function wrapped in a Lambda layer).

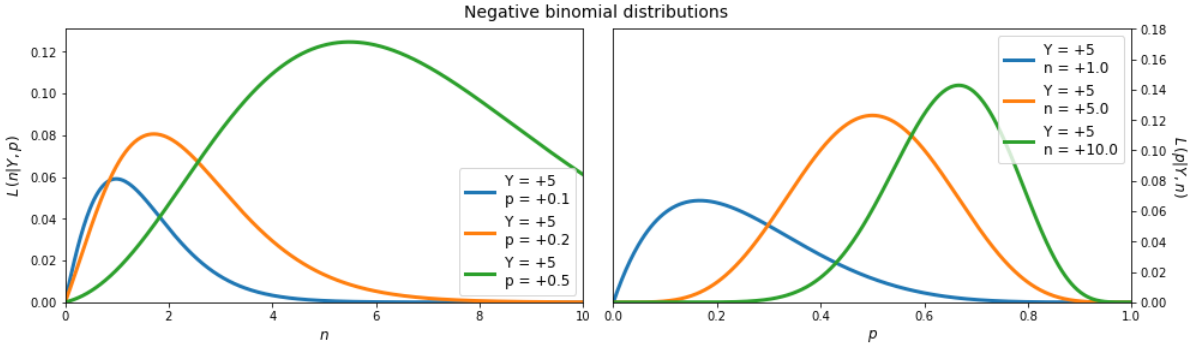
Writing the loss function

By following through the steps detailed above, we have now constructed a model that predicts a set of parameters, θ, which describe the probability distribution of the target variable. However how do we go about training this model? The parameters θ are not simply an estimate of the target variable *Y*, and so we cannot use the same loss metrics as we would for a regression model (e.g. the mean squared error).

Instead we need to revisit our probability distribution $f(y|\theta)$. When training we know the value of the target variable, *Y*, and so we want to find the values of θ that maximise the probability when $y=Y$. Although the equation remains essentially unchanged from the probability distribution, we are now varying the parameters θ for a known value of *Y* — rather than calculating the probability across all values of *y* for a known set of parameters θ. This is called the likelihood, $L(\theta|Y)$. For the negative binomial distribution we are considering, the likelihood is:

$$\mathcal{L}(n,p|Y) = \frac{\Gamma(Y+n)}{\Gamma(Y+1)\Gamma(n)}p^n(1-p)^Y$$

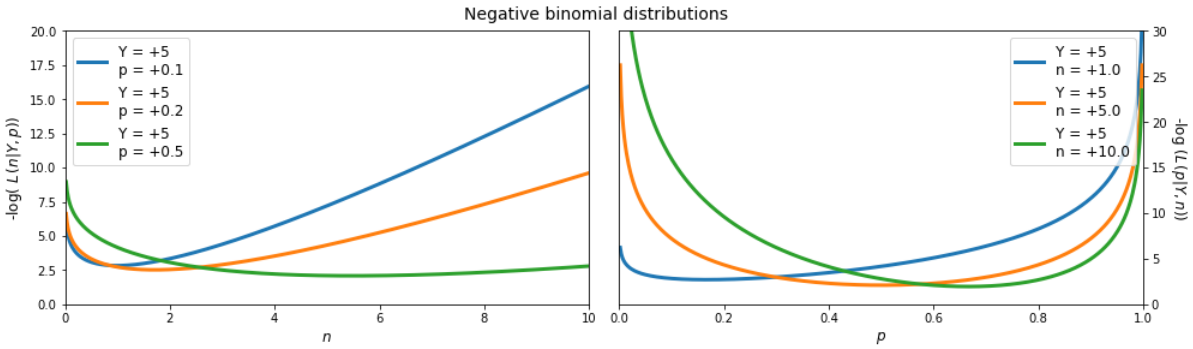
The figures below show some possible shapes of the likelihood for a negative binomial distribution. In each case we're fixing one of the two parameters θ and varying the other.



In all cases there's a clear peak in the likelihood around the optimum value of each parameter, and so we could in theory use the likelihood as the loss function when training our model (with the model aiming to maximise the

loss). However far away from the peak the likelihood is relatively unchanging, and so at the start of training — when the parameters are a long way from their optimal values — the updates to the network will be small.

To overcome this we will use the **negative log likelihood** as our loss function, -log*(L(θ|Y))*. As we are taking the logarithm, the rate of change is large far away from the optimal value and small nearby it. This will help our model converge to the correct values of *θ*. For the sake of tradition we also want to minimise (rather than maximise) the loss function, so we take the negative. Below we plot the negative log of the previous likelihood plots to illustrate this. For a more mathematical justification of why we use the negative log likelihood, take a look here.



For the negative binomial distribution we are using, the negative log likelihood is given by the following (this is simply -log*(L(n,p|Y))*):

$$\mathrm{NLL}(n, p|Y) = \log\Gamma(n) + \log\Gamma(Y+1) - \log\Gamma(Y+n) \\ -n\log(p) - Y\log(1-p)$$

By training our model to minimise this function with respect to *n* and *p*, we will be teaching it to predict, from the inputs *X*, the parameters of the negative binomial distribution which best describes the target variable *Y*. However first we'll need to implement this equation as a custom loss function for Keras to use.

By convention all loss functions used by Keras take two inputs and return one output. The first input, `y_true`, is the target value and the second, `y_pred` is the prediction returned by the model, in our case the predicted values *n* and *p*. The output is the loss. As in our negative binomial layer, we first create separate tensors from *n* and *p* from their single input, and then use these to calculate the negative log likelihood. The logΓ function is already included in `tf.math`, which saves us having to define it.

Note that both the log and logΓ functions are only defined for positive inputs, so if $n \leq 0$, $p \leq 0$ or $p \geq 1$ the above function will return *nan*. However, because we've already fixed *n* and *p* to within these limits by using activation functions, the loss should always be defined.

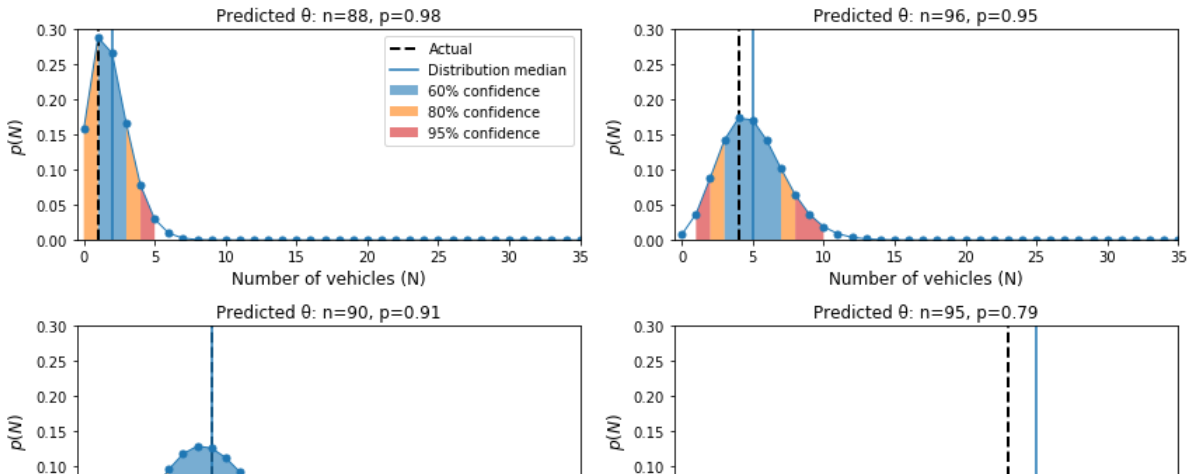
Now that we have defined our loss function, we can compile the model and start training. Don't worry too much about the absolute values of the loss for each epoch: unlike, say, the mean squared error, there is no intrinsic lower bound on the value the negative log likelihood can take. As long as the training and validation curves are decreasing, you can be confident that the model is learning.

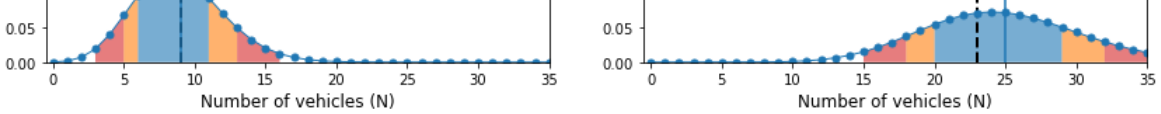
Working with the outputs

Once the training has finished (and assuming the loss has dropped / converged) we will have a model that takes a set of input features, *X*, and returns a set of parameters, *θ*, which describe a probability distribution for the target variable, *Y*. What exactly can we do with these parameters?

For most common distributions (including all those mentioned above) we can make use of the `scipy.stats` module for working with our parameters. This allows us to easily calculate the probability distributions (`.pmf` for discrete distributions, `.pdf` for continuous), expectation value (`.mean`) and confidence intervals (`.interval`), amongst many other things. As we're using a negative binomial distribution, we'll use `scipy.stats.nbinom`.

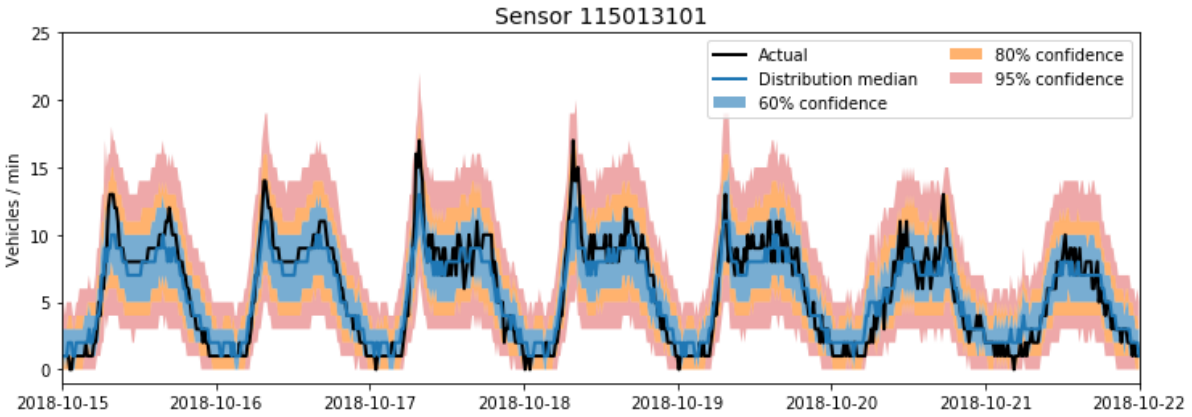
Using the probability mass function, we can easily plot the probability distribution returned by our model for a single sample. This allows us to ascertain whether the model is confident about the value the target variable will take (e.g. the top left plot below), or if it thinks there'll be a large margin of error (e.g. the bottom right plot below).





To reduce our probability distribution to a single prediction of the target, we just have to take the average of the distribution. As we're dealing with count data I have chosen to take the median (to ensure an integer prediction), however it would be equally valid to calculate the mean or mode, depending on the nature of the quantity being predicted. This can then be used to calculate more interpretable accuracy metrics (e.g. mean squared error), and to compare this approach to other, non-probabilistic, modelling methods.

Finally, to return to our initial justification for predicting a probability distribution, we can also generate confidence intervals. Unlike those resulting from the previous, dropout-based approach, these confidence intervals are explicitly learnt from the variation in the training data, and can be generated from a single inference call. This makes them more accurate — note the smoothness of the intervals in the plot below — and much, much faster to predict.



Although it takes a bit more effort to design a neural network that predicts a probability distribution — as opposed to the target variable — if knowledge of the uncertainty is important to the end user then the process is definitely worth pursuing. For many of our customers the uncertainty on our predictions will directly affect the actions they take based upon them, and so we need to ensure our uncertainty estimates accurately reflect the situation at hand.