towardsdatascience.com

Predicting Probability Distributions Using Neural Networks

Shaked Zychlinski

6–7 minutes

This blog post was originally published on [Taboola's Engineering Blog](#)

If you've been following our tech blog lately, you might have noticed we're using a special type of neural networks called Mixture Density Network (MDN). MDNs do not only predict the expected value of a target, but also the underlying probability distribution.

This blogpost will focus on how to implement such a model using Tensorflow, from the ground up, including explanations, diagrams and a Jupyter notebook with the entire source code.

What are MDNs and why are they useful?

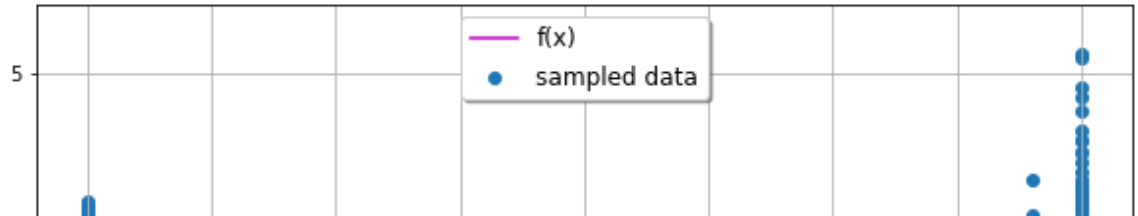
Real life data is noisy. While quite irritating, that noise is meaningful, as it gives a wider perspective of the origins of the data. The target value can have different levels of noise depending on the input, and this can have a major impact on our understanding of the data.

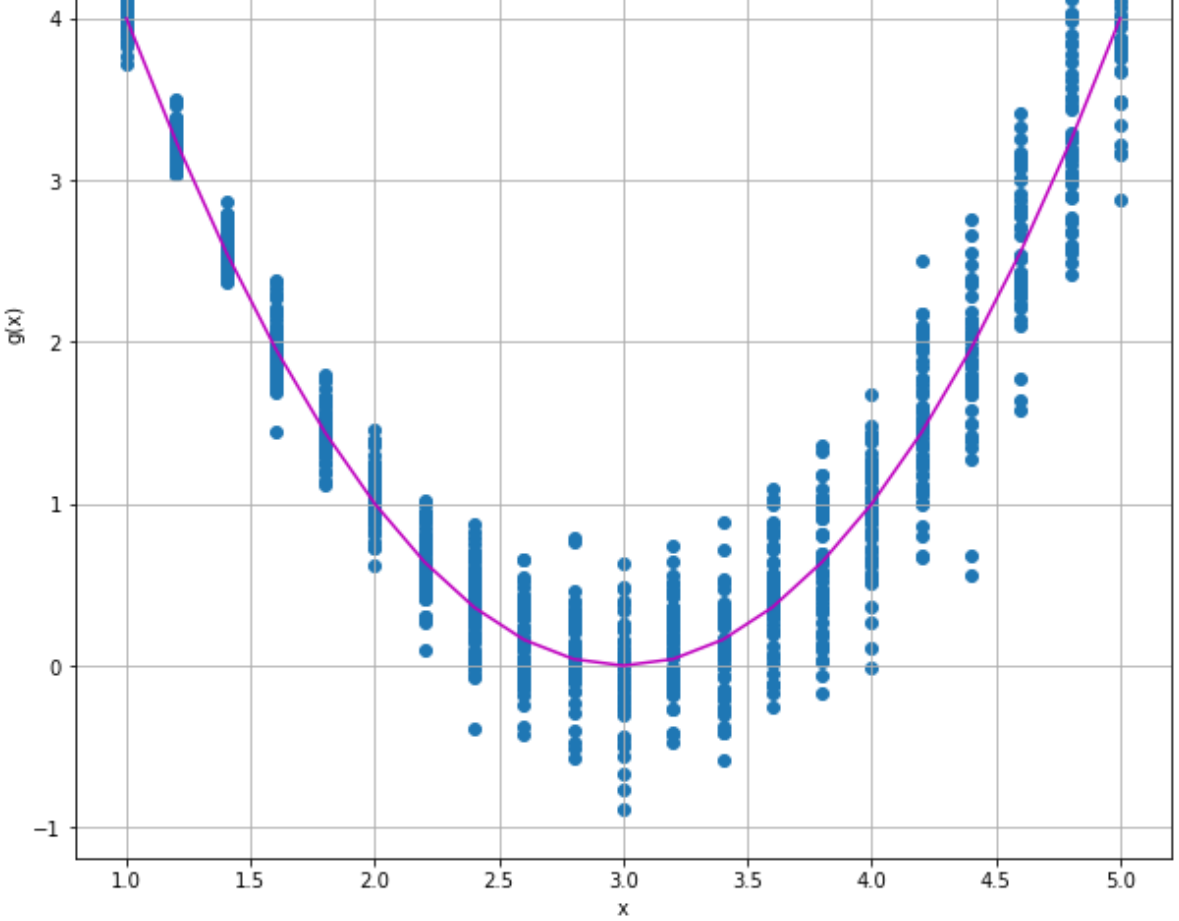
This is better explained with an example. Assume the following quadratic function:

$$f(x) = x^2 - 6x + 9$$

Given x as input, we have a deterministic output *f(x)*. Now, let's turn this function into a more interesting (and realistic) function: we'll add some normally distributed noise to *f(x)*. This noise will increase as *x* increases. We'll call the new function *g(x)*, which formally equals to $g(x) = f(x) + \varepsilon(x)$, where $\varepsilon(x)$ is a normal random variable.

Let's sample *g(x)* for different *x* values:



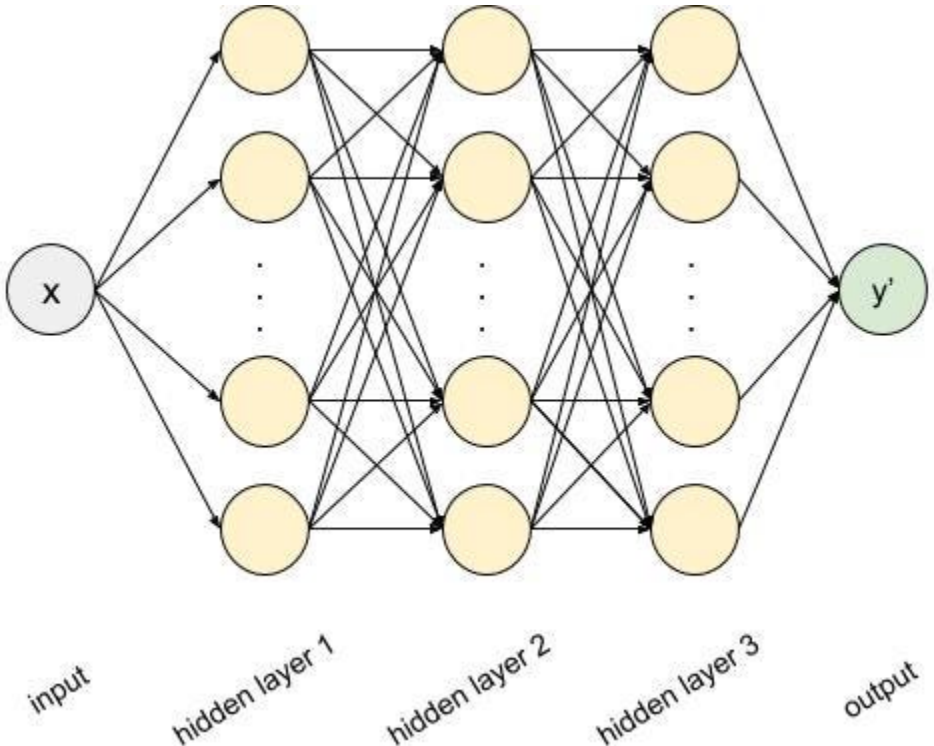


The purple line represents the noiseless function *f(x)*, and we can easily see the increase in the added noise. Let's inspect the cases where *x = 1* and *x = 5*. For both these values, *f(x) = 4*, and so 4 is a reasonable value *g(x)* can take. Is 4.5 a reasonable prediction for *g(x)* as well? The answer is clearly no. While 4.5 seems to be a reasonable value for *x = 5*, we cannot accept it as a valid value for g(x) when *x = 1*. If our model simply learns to predict *y'(x) = f(x)*, this valuable information will be lost. What we actually need here is a model capable of predicting *y'(x) = g(x)*. And this is exactly what an MDN does.

The concept of MDN was invented by Christopher Bishop in 1994. His original paper explains the concept quite well, but it dates back to the prehistoric era when neural networks weren't as common as they are today. Therefore, it lacks the hands-on part of how to actually implement one. And so this is exactly what we're going to do now.

Let's get it started

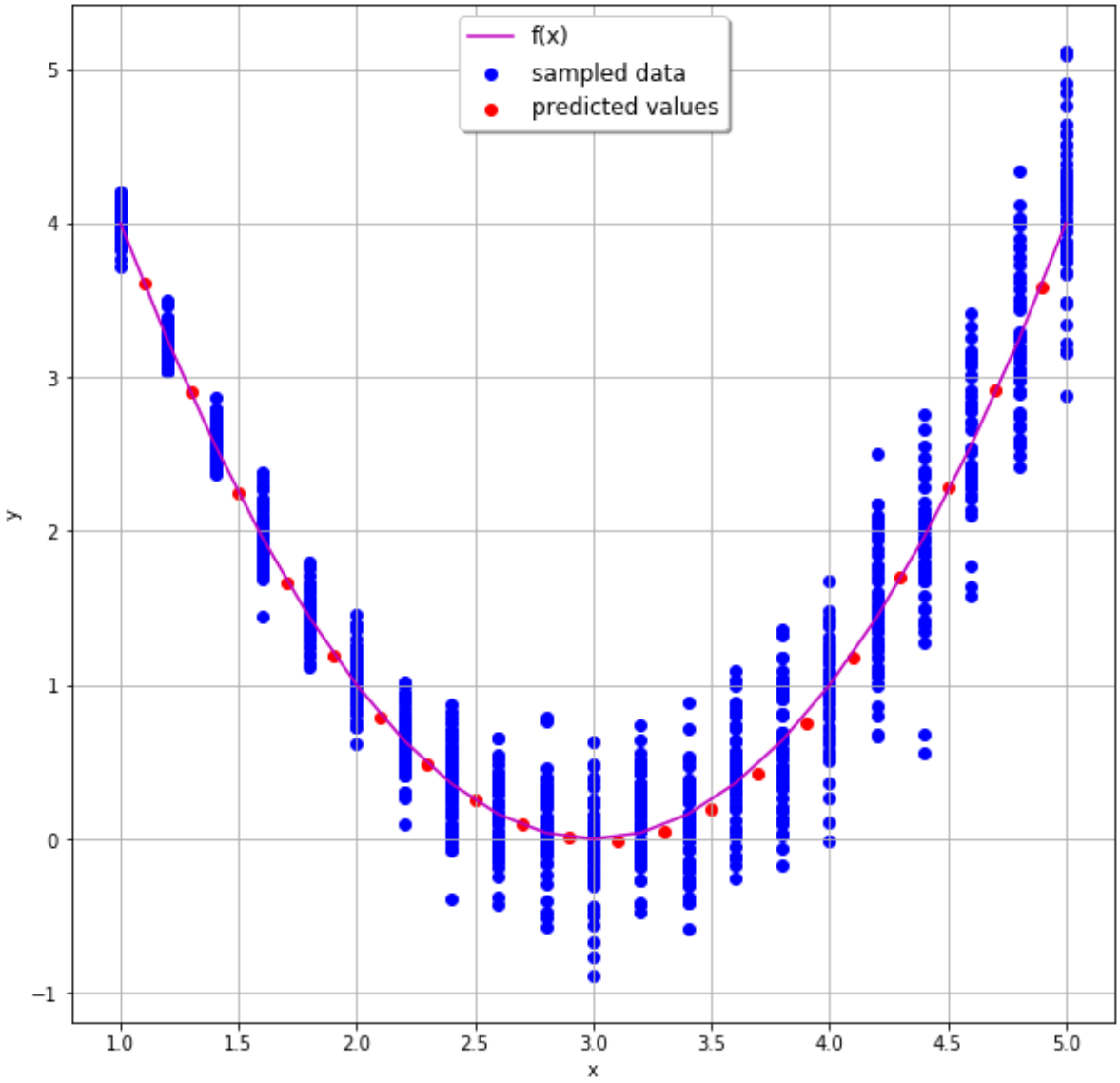
Let's start with a simple neural network which only learns *f(x)* from the noisy dataset. We'll use 3 hidden dense layers, each with 12 nodes, looking something like this:



We'll use Mean Square Error as the loss function. Let's code this in Tensorflow:

```
x = tf.placeholder(name='x',shape=(None,1),dtype=tf.float32)
layer = x
for _ in range(3):
   layer = tf.layers.dense(inputs=layer, units=12, activation=tf.nn.tanh)
output = tf.layers.dense(inputs=layer, units=1)
```

After training, the output looks like this:



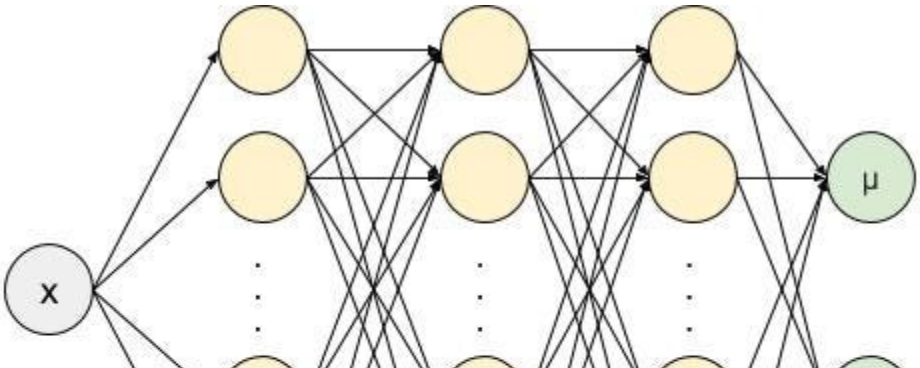
We see the network successfully learnt *f(x)*. Now all that's missing is an estimation of the noise. Let's modify the network to get this additional piece of information.

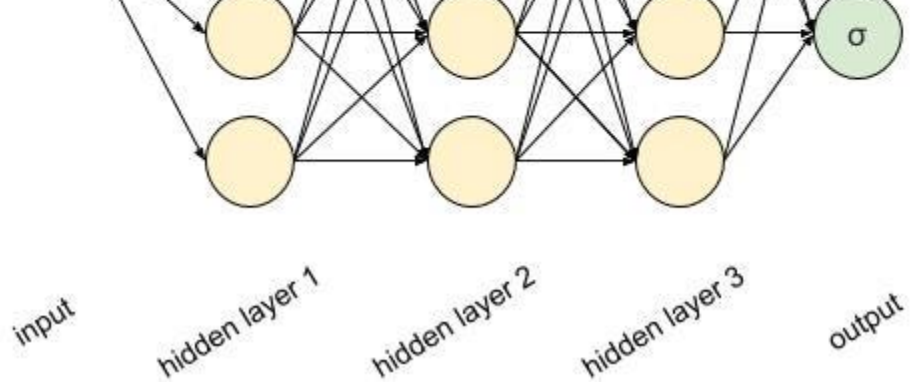
Going MDN

We will keep on using the same network we just designed, but we'll alter two things:

1. The output layer will have two nodes rather than one, and we will name these *mu* and *sigma*
2. We'll use a different loss function

Now our network looks like this:





Let's code it:

```
x = tf.placeholder(name='x',shape=(None,1),dtype=tf.float32)
layer = x
for _ in range(3):
   layer = tf.layers.dense(inputs=layer, units=12, activation=tf.nn.tanh)
mu = tf.layers.dense(inputs=layer, units=1)
sigma = tf.layers.dense(inputs=layer, units=1,activation=lambda x: tf.nn.elu(x) + 1)
```

Let's take a second to understand the activation function of sigma — remember that by definition, the standard deviation of any distribution is a non-negative number. The Exponential Linear Unit (ELU), defined as:

$$ELU(x) = \begin{cases} x & x \geq 0 \\ \exp(x) - 1 & x < 0 \end{cases}$$

yields *-1* as it lowest value, and so *ELU+1* will always be non-negative. Does it have to be ELU? No, any function that always yields a non-negative output will do — for example, the absolute value of sigma. ELU simply seems to be doing a better job.

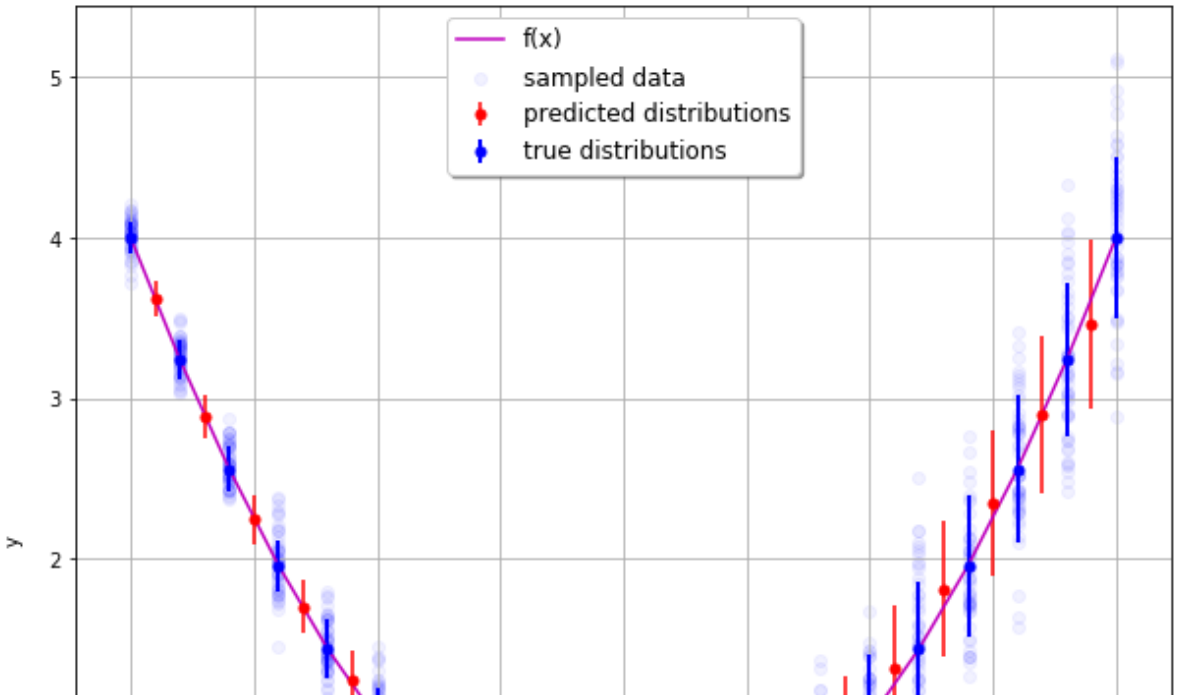
Next, we need to adjust our loss function. Let's try to understand what exactly we are looking for now. Our new output layer gives us the parameters of a normal distribution. This distribution should be able to describe the data generated by sampling *g(x)*. How can we measure it? We can, for example, create a normal distribution from the output, and maximize the probability of sampling our target values from it. Mathematically speaking, we would like to maximize the values of the probability density function (PDF) of the normal distribution for our entire dataset. Equivalently, we can minimize the negative logarithm of the PDF:

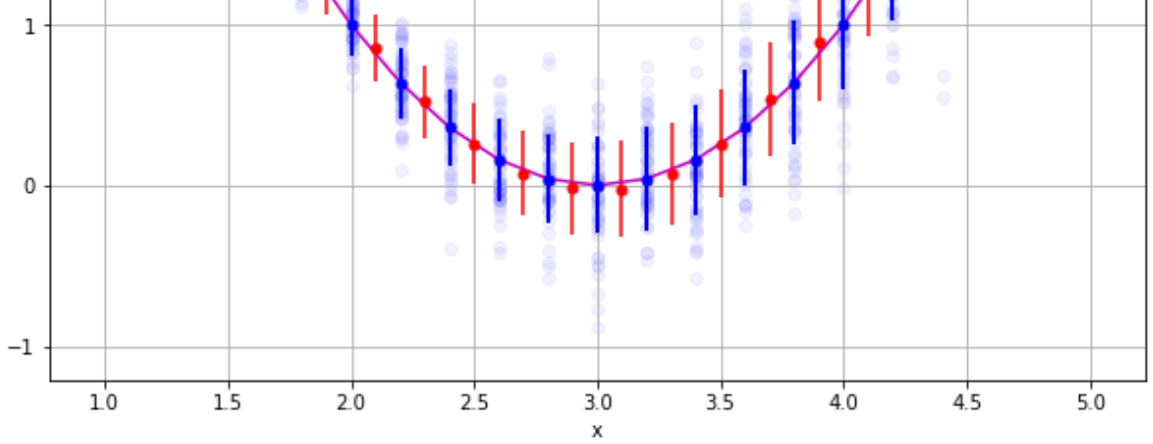
$$Cost = -\log(PDF) = -\log\left(\frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]\right)$$

We can see that the loss function is differentiable with respect to both μ and σ. You'll be surprised by how easy it is to code:

```
dist = tf.distributions.Normal(loc=mu, scale=sigma)
loss = tf.reduce_mean(-dist.log_prob(y))
```

And that's about it. After training the model, we can see that it did indeed pick up on *g(x)*:





In the plot above, the blue lines and dots represent the actual standard deviation and mean used to generate the data, while the red lines and dots represent the same values predicted by the network for unseen *x* values. Great success!

Next steps

We've seen how to predict a simple normal distribution — but needless to say, MDNs can be far more general — we can predict completely different distributions, or even a combination of several different distributions. All you need to do is make sure to have an output node for each parameter of the distribution's variables, and validate that the distribution's PDF is differentiable.

I can only encourage you to try and predict even more complex distributions — use the notebook supplied with this post, and change the code, or try your new skills on real life data! May the odds be ever in your favor.