4

Building loss functions with the likelihood approach

This chapter covers:

- Using the maximum likelihood principle for estimating model parameters
- Using the maximum likelihood principle to determining a loss function for classification problems
- Using the maximum likelihood principle to determining a loss function for regression problems

Deep Learning models have often millions of parameters which you need to determine during the training process. In chapter 3 you have seen how you can determine the parameter values via optimizing a loss function via stochastic gradient descent (SGD).

But how did we arrive at the loss function? In the linear regression problem, we used the mean squared error as a loss function. We don't claim that it is a dumb idea to minimize the squared distances of the data points from the curve. But why use squared and not, for example, the absolute differences? In chapter 2, we considered a classification problem where the task was to decide if a banknote was faked or not, or in the other example, we classified an image of a handwritten digits to be either 0,1,...,9. In those cases we used a loss function called categorical cross entropy. What is this and how do we get to it in the first place?

In this chapter, we introduce the maximum likelihood principle and show how to use it to determine an appropriate loss function for the classification and regression problems. In the next chapter, we use the maximum likelihood for other problems. As it turns out the maximum likelihood principle is behind almost all loss functions used in deep learning.

4.1 The maximum likelihood principle

So what is this maximum likelihood principle which is so important in deep learning?

We start with a simple example far away from deep learning. Consider a die with one face showing a dollar sign ($) and the remaining five faces are dots (see figure 4.1).

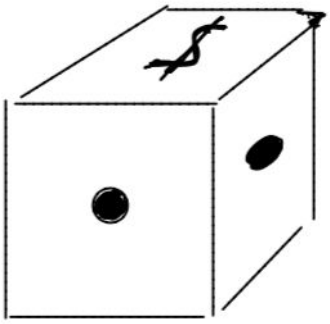

Figure 4.1 A die with one side showing a dollar sign and the others a dot.

So what is the probability that you get the dollar sign up if you throw the die (we assume fair die here)? On average, you will get a $ sign up in one out of six cases. The probability to see a dollar sign is thus *p*=1/6. The probability that this won't happen and you see a dot is 5/6 = 1-*p*. Let's throw the die10 times. What is the probability that you see the dollar sign only one time and the dot nine times? Let's assume first, that you see the dollar sign in the first throw and you see dots in the next nine throws. You could write this in a string as:

```
$.........
```

The probability for that particular sequence to happen would be ⅙ * ⅚ * ⅚ * ⅚ * ⅚ ...⅚ = ⅙*⅚^9=0.032 or with *p*=⅙ as *p*^1 * (1-*p*)^(10-1). If we only ask for the probability that a single dollar sign and nine dots occurred in the ten throws regardless of the position, we have to take all of the following ten results into account

```
$.........
.$........
..$.......
...$......
....$.....
.....$....
......$...
.......$..
........$.
.........$
```

Each of these ten distinct sequences have the same probability of $p \cdot (1-p)^9$ to occur. This yields a probability of $10 \cdot p \cdot (1-p)^9$ for the event that one out of these ten sequences is observed. In our example we have *p*=⅙ and we get 0.323 for the probability that one dollar sign occurred during the ten throws. You might get curious and ask yourself what is the probability for two dollar signs in ten throws. The probability for a particular ordering (say `$$........`) would be $p^2 \cdot (1-p)^8$. It turns out that there are now 45 possible ways[1] to rearrange the string like `$.$.......` or `$..$.......` the total probability of two dollar signs and eight dots is thus: $45 \cdot (⅙)^2 \cdot (⅚)^8 = 0.2907$.

The dice throwing experiment, where we count the successful throws with upcoming $, is an example of a general class of experiments that are called binomial experiments. In a binomial experiment you count the number of successes (here seeing a $) in n trials (here throwing a die) where all trials are independent from each other and the probability for success is the same in each trial. The number of successes in a binomial experiment with *n* trials is not fixed but can usually take any values between 0 and *n* (if *p* is not exactly 0 or 1). For this reason the number of success *k* is called a *random variable* for which we can derive a probability for each possible outcome which can be summarized in a binomial distribution (see figure 4.2). There is a scipy function to calculate it called bimom.pdf with the argument k=`number of successes`, n=`number of tries` and p=`probability for success` using this function, we can plot the probability that 0 to 10 dollar signs occur in 10 throws. See listing 4.1 for the code and figure 4.2 for the result.

[1],

Listing 4.1 The probability for 0 to 10 red throws using the binom function

```
from scipy.stats import binom
ndollar = np.asarray(np.linspace(0,10,11), dtype='int') #A
pdollar_sign = binom.pmf(k=ndollar, n=10, p=1/6)  #B
plt.stem(ndollar, pdollar_sign)
plt.xlabel('Number of dollar signs')
plt.ylabel('Probability')

#A The numbers of successes (thrown dollar sings) 0 to 10 as integers
```

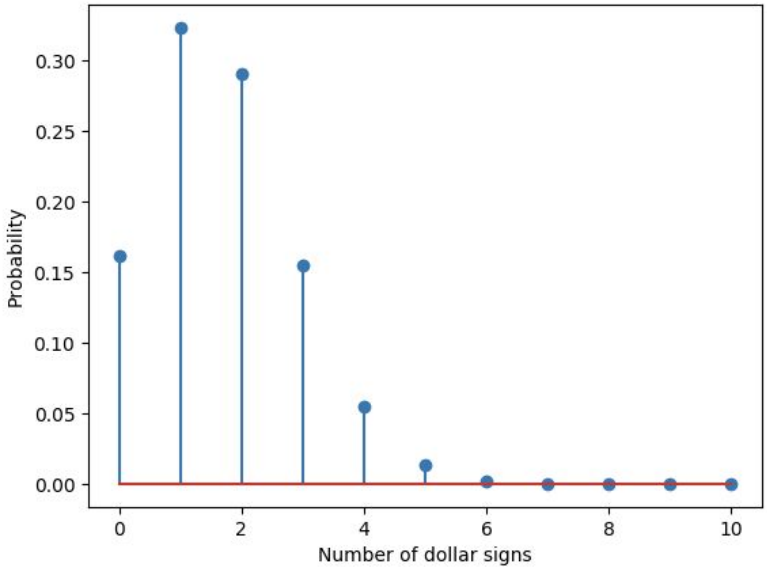

Figure 4.2 Probability distribution for the number of observed dollar signs in 10 dice throws. The probability of a dollar sign in an individual throw is p=⅙. The figure is created with Listing 4.1. The probability for one and two dollar signs (0.323 and 0.2907) is the same as calculated by hand.

So far so good, you might remember this from your probability class. Now we turn the tables. Consider the following situation: You are in a casino and play a game in which you win, if the dollar sign appears. You know that there are certain number of faces (0 to 6) with the dollar sign, but you don't know how many. You observe 10 throws of the dice and 2 dollar signs came up in that throws. What do you guess for the number of dollar signs on the dice? Sure it cannot be zero since you observed a dollar sign on the other hand it cannot be 6 either, you observed dots. But what would be a good guess? Looking at the Listing 4.1 again, you suddenly have a genius idea. Simply calculate the probabilities to observe two dollar signs in ten throws again, but this time assume that your die has not only on one face with a dollar sign but on two faces. Then assume your die has on three faces a dollar sign and again determine the probability to see two dollar signs in ten throws, and so on. So, your observed data is fixed (two dollar signs in ten throws) but your assumed model of how the data is generate changes from a dice with zero dollar faces to a dice with 1, 2, 3, 4, 5, or 6 dollar faces. The probability to observe of a dollar sign in one throw can be seen as parameter in your model of the di and this parameter value take the values $p = \frac{0}{6}, \frac{1}{6}, \frac{2}{6}, ..., \frac{6}{6}$ for your different die models. For each of these dice models you can determine the probability to

observe two dollar signs in ten throws and plot it in a chart (see figure 4.3).

Hands-on time: Open the notebook https://github.com/tensorchiefs/dl_book/blob/master/chapter_04/nb_ch04_01.ipynb and work through the code until you reach the first exercise. In the exercise it is your task to determine the probability to observe two-times a dollar sign in ten dice throws, if you consider a die that has dollar signs on 0, 1, 2, 3, 4, 5, or all 6 faces. Plotting the computed probabilities yields the plot in figure 4.3.

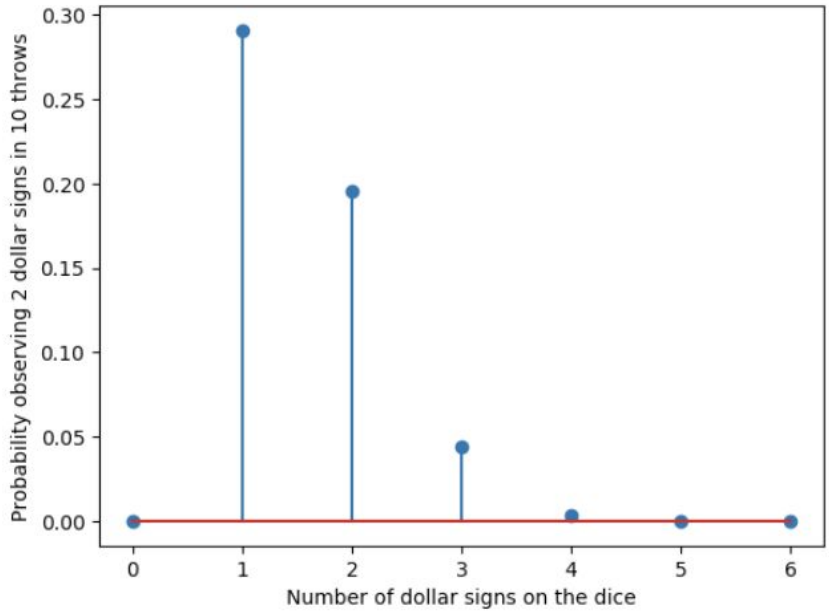

Figure 4.3 Likelihoods for observing *k*=2 dollar signs in *n*=10 throws for different number of dollar signs on the die.

What do we see in Figure 4.3? Starting from the very left. If you have zero dollar signs on the die then the probability that you observe two dollar signs in ten throws is zero. Well that was expected. Next compute the probability to observe two times a $ in ten throws assuming that you have only one dollar sign on the die (p=⅙) and this is nearly 0.3. If you assume a die with two dollar signs then the probability to observe two $ in ten throws is approximately 0.20 and so on. So what would you guess is the number of dollar signs on the die? You would guess one, since a die with one $ side has the highest probability to yield two $ in ten throws. Congratulation you have just discovered the maximum likelihood principle.

The Maximum Likelihood Principle Mantra:

Choose the parameter(s) of the model so that the observed data has the highest likelihood

In our case, the parametric model is the binomial distribution, which has two parameters: the success probability *p*

per trial and the number of conducted trials n. We observed the data: k=2 dollar signs at n=10 throws. The parameter of the model is p, the probability that a single dice throw shows a dollar sign. The likelihood for different number of dollar signs on the die is shown in Figure 4.3. We choose the value with maximal likelihood (p=⅕), corresponding to one dollar sign on the dice.

There is a small subtlety, the probabilities in Figure 4.3 do not sum up to 1, but to 0.53. Therefore, the probabilities are not probabilities in the strict sense which must sum up to 1. This is the reason, we speak of likelihoods instead of probabilities. But we can still use these likelihoods for ranking and we pick the model yielding the highest likelihood as the most probable model.

Let's recap the steps in this maximum likelihood approach for determining the best parameter values:

1) You need a model for the probability distribution of the observed data which has one or several parameters
    - here the data was the number of times you saw the dollar sign when throwing the die ten times and the parameter was the number of $ faces of that diee
2) You use the model to determine the likelihood to get the observed data when assuming different values of the parameter in the model
    - Here you computed the probability to get two dollar signs in ten throws when your assumed dice has either 0, 1, 2, 3, 4, 5, or 6 dollar faces.
3) You take the parameter value for which the likelihood to get the observed data is maximal as the optimal parameter value - also called Maximum Likelihood (ML) estimator.
    - here the ML estimator is that the die has one side with a dollar sign.

4.2 Deriving a loss function for a classification problem

In this section, we take a closer look at the loss function for classification problems. Let's revisit the fake banknote example from chapter 2 again. There were two versions of neural networks, one with a single output neuron and one with two output neurons. Let's first look at the version with two output neurons (see figure 4.3a). In this case a loss function called 'categorical_crossentropy' (see Listing 2.2). This 'categorical_crossentropy' was also used for the remaining classification problems of chapter 2, the fully connected network, the CNN applied to the MNIST handwritten digits classification problem, and the CNN for detecting stripes in an art pieces. This loss function is generally used in classification problems.

4.2.1 The loss function for classification in networks with one output per class

We start with the banknote example. In that case the output of the first neuron (labeled p0 in figure 4.3a) is the probability that the model "thinks" a given input x belongs to class 0 (a real banknote). The output of the other neuron p_1 is the probability that x describes a fake class. Of course p_0 and p_1 add up to one, this is assured by the softmax activation function, see chapter 2.

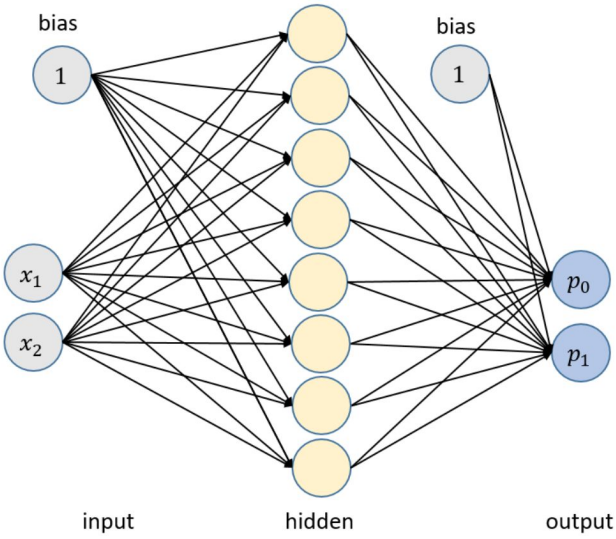

Figure 4.3a The classification network for banknotes described by to features x_1 and x_2 with two outputs yielding the probability for a real banknote (p_0) or a fake banknote (p_1). Same as figure 2.8 in chapter 2.

We now use the maximum likelihood principle to derive the loss function. What is the likelihood of the observed data? The training data in classification problems comes in pairs (x_i, y_i). In the banknote examples x_i is a vector with two entries and y_i is the true class of the example (banknote is fake or real). In classification we keep x_i fixed and look at the probability of the true class y_i given x_i. For a NN with known weights and a given input x_i you can directly compute for the resulting $p_0(x_i)$ for $y_i = 0$ or $p_1(x_i)$ if $y_i = 1$.

To keep in mind: the likelihood of a classification model for a given training example (x_i, y_i) is simply the probability the network assigns it to the correct class y_i.

As an example, a well trained network would have a high value for p_1 if the training example is from class 1 (fake banknote). What is probability for the training set as a whole? Here we make the assumption that all the examples in the training are independent of each other. Then the probability of the whole training data set is simply the product of the individual probabilities. For example: Imagine you throw a regular dice twice and you ask for the probability to have a one in the first throw and a two in the second. This is simply $\frac{1}{6} \cdot \frac{1}{6}$ since the throws are independent. So the likelihood of the whole training set is the product of all individual examples. So we go one training example after another and look what is the probability for the true class and then take product of all those probabilities. We could also order the probabilities as follows: First we take the real banknotes (for which y=0) and multiply the predictions p_0 together. Then, we take the fake ones and multiply p_1 together. This can be written as:

$$Pr(Training) = \prod_{j\ for\ with\ y=0} p_0(x_j) \cdot \prod_{j\ for\ with\ y=1} p_1(x_j) \qquad (1)$$

Equation (1) depends on the number *n* of training examples and will increase with the amount of the data. To judge the likelihood value it is easier to look at the mean likelihood per observation which you can easy obtain by dividing the equation (1) by n. You are allowed to do so, because dividing with constant factor does not change the position of the maximum.

Equation (1) can also be explained in a slightly different way which is based on formulating the probability distribution for an outcome Y. Because this point of view can help you to digest the ML approach from a more general view we give this explanation in the sidebar "ML approach for the classification loss using a parametric probability model".

Sidebar: ML approach for the classification loss using a parametric probability model

The NN with fixed weights in figure 4.3a outputs the probabilities for all possible class labels y when feeding in a certain input x. This can be written als probability distribution for the outcome Y which depends on the input value x and the weights of the NN.

$$P(Y=k \mid X=x, W=w) = \begin{cases} p_0(x,w) & \text{for } k=0 \\ p_1(x,w) & \text{for } k=1 \end{cases} \quad \text{with } \sum p_i = 1$$

$Y=k$ states that the random variable Y takes specific values k. You can further read the vertical bar as "with given" or "conditioned on" and behind the bar comes all information which is given and is used to determine the probability of the variable in front of the bar. Here you need to know the value of the input x and the weights w of the NN to compute the outputs of the NN which are p0 and p1. Because the probability distribution depends on x, it is called a conditional probability distribution (CPD). Often you see a simpler version of this equation which skips the part behind the bar (either only $W=w$ or both $X=x,\ W=w$) and assumes that this is self-evident.

$$P(Y=k) = \begin{cases} p_0 & \text{for } k=0 \\ p_1 & \text{for } k=1 \end{cases} \quad \text{with } \sum p_i = 1$$

This probability distribution, where the outcome *Y* can only take the value 0 or 1, is known as Bernoulli distribution, which has only one parameter p_1 since from this you can directly compute $p_o = 1 - p_1$.

This probability distribution for the binary outcome Y is visualized in figure 4.3b.

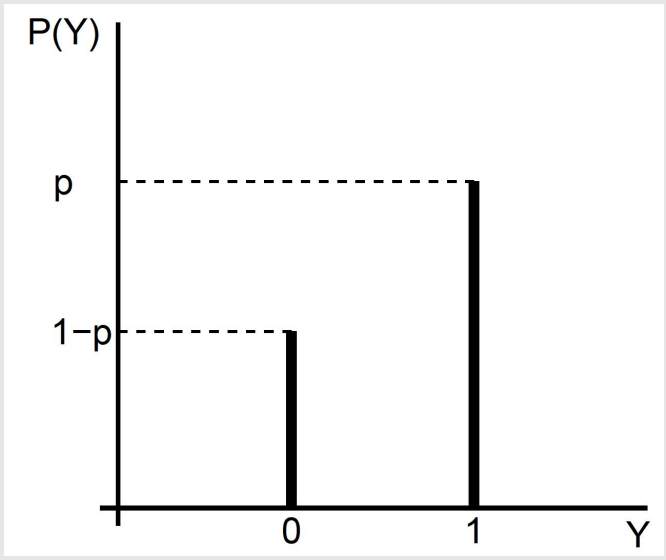

Figure 4.3b The probability distribution of a binary variable Y, also known as Bernoulli distribution

The NN in figure 4.3a computes for each input p_0 and p_1. For n data points the probability or likelihood of the data is given by the product of the computed probabilities for the correct class - see equation (1). The maximum likelihood principle tells you that you should tunee network weights $w=(a,b)$ such that the likelihood gets maximized:

$$(a,b) = argmax_{(a,b)} \left\{ \prod_{i=1}^{n} P(Y = y_i | x_i, a, b) \right\}$$

$$= argmax_{(a,b)} \left\{ \prod_{i \text{ for which } y_i=0}^{n} P(Y = 0 | x_i, a, b) \cdot \prod_{i \text{ for which } y_i=1}^{n} P(Y = 1 | x_i, a, b) \right\}$$

Remark: In case of more than two possible classes you would use a NN with as many output neurons as classes. Each output represents then the probability of one of the possible classes. Using the softmax activation function in the output layer guarantees that these probabilities are non-negative and add up up to one. The corresponding probability distribution is the called Multinomial distribution. We come back to the case of more than 2 classes later in the chapter (see equation 4).

In principle we are now done. We could maximize Equation 1 by tuning the weights of the network. You do not need to do this by hand, but can use any framework, such as tensorflow or keras, that can do (stochastic) gradient descent.

There is still a practical issue. The p_0 and p_1 in Equation 1 are numbers between 0 and 1, some of them might be small. Multiplying many numbers in the range 0 to 1 yield to numerical problems (see Listing 4.2).

Listing 4.2 Numerical instabilities when multiplying many numbers between 0 and 1

```
import numpy as np

vals100 = np.random.uniform(0,1,100)#A

vals1000 = np.random.uniform(0,1,1000) #B

x100 = np.product(vals100)

x1000 = np.product(vals1000)

x100, x1000 # C

# A multiplying 100 numbers randomly drawn between 0 and 1.0

# B multiplying 1000 numbers randomly drawn between 0 and 1.0

# C (7.147335361549675e-43, 0.0) a typical result: for 1000 numbers you get zero
```

If we take more than a few hundred examples the product is so close to 0 that it is treated as zero due to the limited precision of the float numbers in a computer. In deep learning typical float numbers of type float32 are used and for those the smallest number next to zero is approximately 10^-45.

There is a trick to fix this issue. Instead of Pr(Training) in Equation 1, you can take the logarithm of the likelihood. Taking the logarithm changes the values of a function but does not change the position at which the maximum is reached. As a side note: the property that the maximum stays the same, is due to the fact that the logarithm is a strictly *monotonic* functions. In Figure 4.4 you see an arbitrary function f(x) and its logarithm log(f(x)), the maximum of both f(x) and log(f(x)) is reached at $x \approx 500$.

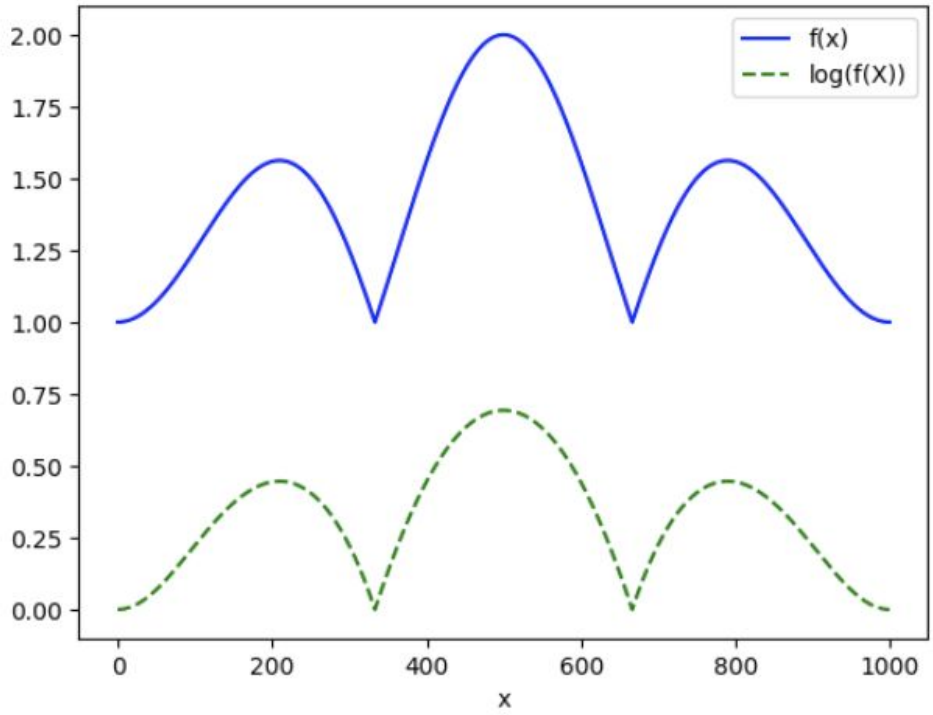

Figure 4.4 A arbitrary function f(x) with non-negative values (solid line) and the logarithm of that function (dashed line). While the maximal value (approximately 2) changes if we take the logarithm, the position at which the maximum is reached (approx 500) stays the same regardless if the logarithm is taken or not.

But what have you gained by taking the logarithm? The log of the product of any numbers is the sum of the logs, which means $\log(A \cdot B) = \log(A) + \log(B)$. This formula can be extended to an arbitrary number of terms $\log(A \cdot B \cdot C \cdot \ldots) = \log(A) + \log(B) + \log(C)$ and so the products in Equation (1) become sums of the logarithms. Let's look at the consequences for the numerical stability. You now add the log of numbers between 0 and 1. For 1 you get log(1)=0, for 0.0001 you get $\log(0.0001) \approx$-4. The only numerical problem would arise if you really get a probability of 0 for the correct class, than the log of zero is minus infinity. To prevent this sometimes a small number like 10E-20 is added to the probabilities. Let's not bother here about this extremely unlikely situation. So what happens if you change Listing 4.2 from products to sum of logs? You get a numerical stable result, see listing 4.3.

Listing 4.3 Fixing the numerical instabilities by taking the log

```
import numpy as np

log_x100 = np.sum(np.log(vals100)) #A

log_x1000 = np.sum(np.log(vals1000))

log_x100, log_x1000 # B

# A The product, based on the same sampled numbers as in Listing 4.2 now becomes the sum of the
logs

# B (-89.97501927715012, -987.8053926732027) a typical result: for 1000 numbers you get a valid
value
```

Doing the log-trick transforms the maximum likelihood equation (1) to the maximum log likelihood equation

$$\log(Pr(Training)) = \sum_{j\ for\ y=0} \log(p_0(x_j)) + \cdot \sum_{j\ for\ y=1} \log(p_1(x_j)) \qquad (2)$$

This log-trick does not depend on the bases you work with, but it is useful to know that Keras uses the natural logarithm to calculate the loss function. We are now nearly at the end, just two minor details. As discussed after equation (1), it is easier to look at the mean likelihood per observation which you can easy obtain by dividing the equation (2) by n. The last point is: Instead of maximizing, deep learning frameworks usually are built to minimize a loss function. So instead of maximizing log(Pr(Training)) you minimize -log(Pr(Training). And voila, you have derived the negative log likelihood function, which is also called *cross entropy!* After achieving our goal to derive the loss function of a binary classifier, let's write it down once more:

$$crossentropy = -\frac{1}{n}\left(\sum_{j\ for\ y=0} \log(p_0(x_j)) + \sum_{j\ for\ y=1} \log(p_1(x_j))\right) \qquad (3)$$

You can now verify that this is indeep the quantity which gets minimized in deep learning.

Hands-on time: Exercise: Open the notebook https://github.com/tensorchiefs/dl_book/blob/master/chapter_02/nb_ch02_03.ipynb. Run the code where you simulate a dataset of 500 images with horizontal edges and 500 images with vertical edges. Your task in the first exercise is to use the function model.evaluate to determine the crossentropy loss of the untrained model and explain the value that you obtain.

With the untrained model you achieve a crossentropy of around 0.7. Of course this is a random result since the weights of the network have their initial random values and no training happened yet, so expect some random deviation from 0.7. Can you explain that value of 0.7 with what you just learned? The networks knows nothing at the beginning. What mean hitting rate would you expect? About 50%, right? Let's calculate $ln(0.5)$. That's 0.69, which looks about right! What mean hitting rate would you expect? About 50%, right? OK, let's calculate $ln(0.5)$. That's 0.69, which fits to the value you get from keras! (Keras uses the natural logarithm to calculate the loss function).

What happens if you have more than two classes? Say you have K classes numbered from 0 to K-1. In equation (3) the sums are then simply taken over the respective classes yielding:

$$crossentropy = -\frac{1}{n}\left(\sum_{j\ for\ y=0} \log(p_0(x_j)) + \sum_{j\ for\ y=1} \log(p_1(x_j)) + \ldots + \sum_{j\ for\ y=K-1} \log(p_{k-1}(x_j))\right) \qquad (4)$$

What do you expect as the starting value for the MNIST example. Pause for a second and try to find the answer by yourself before you read on…

For the MNIST example, we have 10 classes. A non trained network would assign 1/10 to each class. This will lead to p_i = 1/10 for all classes and we get -log(1/10) for the loss which is approximately 2.3. It's a good practice to always check this number for classification problem to debug your training.

Hands-on time: Openthe notebook https://github.com/tensorchiefs/dl_book/blob/master/chapter_04/nb_ch04_02.ipynb and work through the code until you reach the exercise, indicated by a pen icon. Your task in the exercise is to use the digit predictions on MNIST images which are made by an untrained CNN and compute the value of the loss by 'hand'. You'll see that you get a value close to the value 2.3 which you calculated above by hand.

Sidebar: Loss function for classification of two classes with a single output

For the special case that you have 2 classes, like in the banknote example, there is the possibility of having an network with one output neuron. In that case the output is the probability for class p1. The probability for the other class p0 is then given by p0=1-p1. In that case we do not need a one-hot encoding for y_i , it is either $y_i = 0$ for class 0 or $y_i = 1$ for class 1. Using that we can rewrite (3) as:

$$crossentropy = -\frac{1}{n}\left(\sum_{j=1}^{n} (y_i \log(p_1(x_i)) + (1 - y_i)\log(1 - p_1(x_i))\right) \quad (3)$$

In contrast to equation (4), we don't have to check, to which class an example belongs. If the example i belongs to class 1 $y_i = 1$, the first part is taken containing *p*1. Otherwise, if the example i belongs to class 0 $y_i = 0$, the second part with *p*0=1-*p*1 is active.

4.3 Deriving a loss functions for regression problems

In this section, you will use the Maximum Likelihood principle to derive the loss function for regression problems. You will start of by revisiting the blood pressure example from chapter 3, where the input was the age of an american healthy woman and the output was a prediction of her systolic blood pressure (sbp). In chapter 3 you have used a simple NN without hidden layer to model a linear relationship between input and output. As loss function you have used the mean square error. This choice of the loss function was explained by some handwaving arguments, but no hard facts were given to prove that this loss function is a good choice. In this chapter you will see that the MSE as loss function directly results from the maximum likelihood principle.

4.3.1 Using a NN without hidden layer and one output neuron for modeling a linear relationship between input and output

Let's go back to the blood pressure example from chapter 3. In this application you used a simple linear regression model $\hat{y} = a \cdot x + b$ to estimate the systolic blood pressure *y* of a woman when given the age *x* of the woman. Training or fitting this model requires to tune the parameters *a* and *b* so that the resulting model " best fits" to the observed data. In figure 4.5 you can see the observed data together with a linear regression line which goes quite well through the data but might not be the best model. In chapter 3 you have used the mean square error (MSE) as loss function that quantifies how bad the model fits the data (recall equation 3.1):

$$Loss = MSE = 1/n \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = 1/n \sum_{i=1}^{n} (y_i - (a \cdot x_i + b))^2$$

This loss function relies on the idea that deviations between model and data should be quantified by summing up the squared residuals. Given this loss function you have determined the optimal parameter values by stochastic gradient descent (SGD).

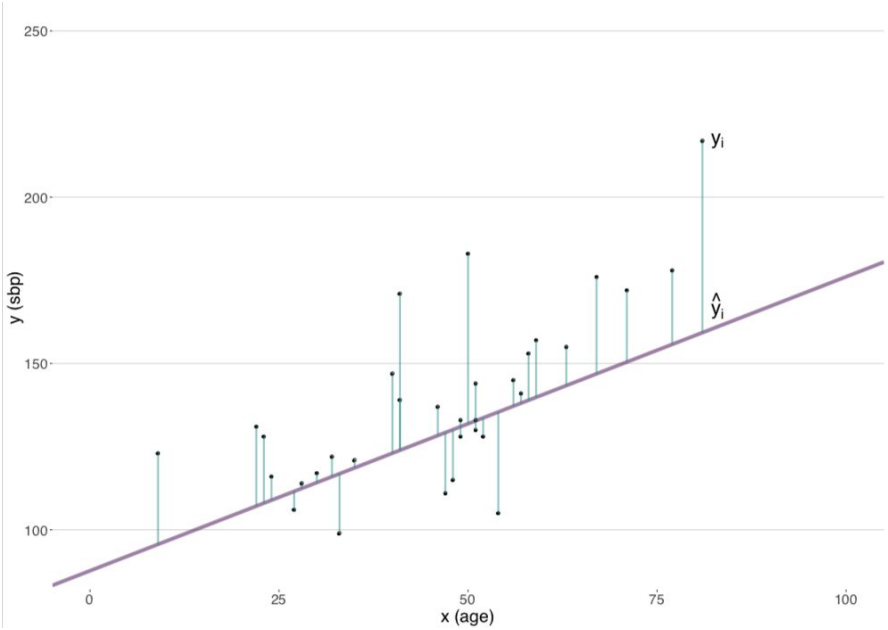

Figure 4.5 Scatterplot and regression model for the blood pressure example: The dots are the measured data points, the straight line is the linear model, the vertical differences between data points and model are the residuals

In chapter 3 we introduced the MSE loss with the quiet hand waving idea that a fit is optimal if the sum of the squared residuals is minimal. In the following you will see how you can use the maximum likelihood principle to derive the appropriate loss for a linear regression task in a theoretical sound way.
Spoiler alert: You will see that the ML approach leads to the MSE loss.

Let's derive the loss function for the regression task by the maximum likelihood approach. For a simple regression model we need only a very simple neural network shown in figure 4.6 without a hidden layer. When using a linear activation function this NN encodes the linear relationship between the input x and the output $out = a \cdot x + b$.

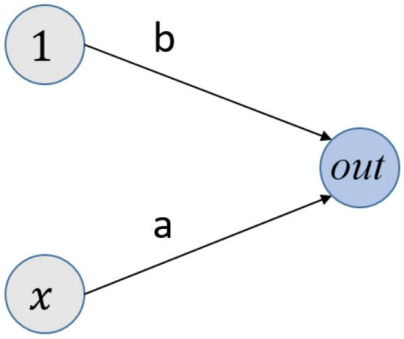

Figure 4.6 Simple linear regression as fully connected neural network without hidden layer computing the output directly from the input as $out = a \cdot x + b$.

The training data in regression problems comes in n pairs (x_i, y_i). In the blood pressure example x_i holds the age of the i-th woman and y_i is the true systolic blood pressure of the i-th woman. If you have chosen certain numbers for the weights of the NN, say $a = 1$ and $b = 100$, then you can compute for a given input, say $x = 50$, the fitted value $\hat{y} = 1 \cdot 50 + 100 = 150$ which you can understand as best guess of the model. In our data set we have a woman with age 50, but her blood pressure is 183 not 150. This does not imply that our model is wrong or could be further improved, because you do not expect that all woman with the same age have the same blood pressure.

As in classification also in regression, the output of the NN, is not the value y you expect to really observe when the input has a specific value x : In classification the output of the NN is not a class label, but the probabilities for all possible class labels which are the parameter of the fitted probability distribution (see figure 4.3b). In regression the output of the NN is not the specific value y, but again the parameter(s) of the fitted continuous probability distribution, which is a Normal distribution.

To recap properties of the Normal distribution please see the sidebar.

Sidebar: Recap on Normal distributions

Let's recap how to deal with a continuous variable Y that follows a Normal distribution. First have a closer look at the density of a Normal distribution $N(\mu, \sigma)$ - the parameter μ determines the center of the distribution and σ determining the spread of the distribution (see figure 4.8). A variable Y that follows a normal distribution,

$Y \sim N(\mu, \sigma)$
has the following probability density function

$$f(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

which is visualized in figure 4.8:

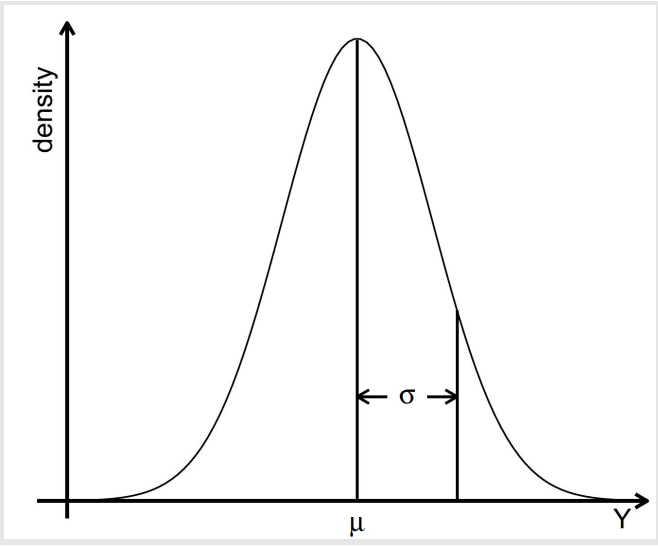

Figure 4.8 The density of a Normal Distribution where μ is the center and σ is the spread of the distribution

Looking at figure 4.8 gives the intuition that Y takes with high probability values close to μ and with small probability values far away from μ. This intuition is correct. But still it is harder to read the probability distribution of a continuous variable Y (see figure 4.8) than the probability distribution of a discrete variable (see figure 4.2). In case of a discrete variable, the probability distributions consists of separated bars, corresponding to the discrete outcome values, where the height of the bars directly correspond to their probabilities and these probabilities add up to one. A continuous variable can take infinitely many possible values and therefore the probability for exactly one value like 3.14159265359... is zero. A probability can therefore only be defined for a region of values. The probability to observe a value y in the range between *a* and *b*, $y \in [a,b]$, is given by the area under the density curve between *a* and *b* (see figure 4.9). The range of all possible values has a probability of one, therefore the area under a probability density is always one.

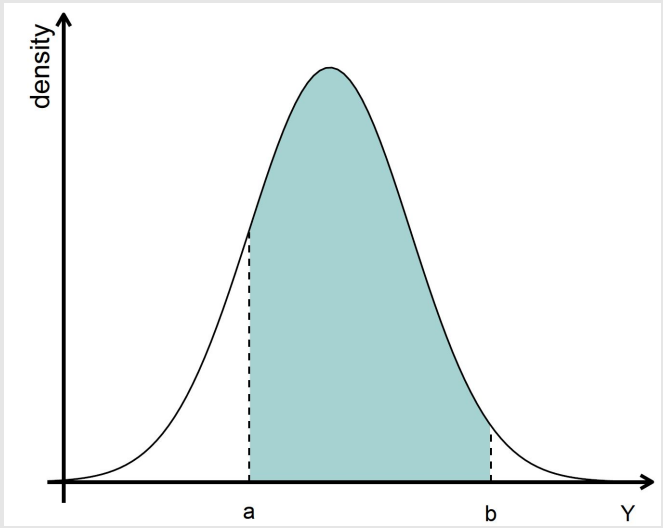

Figure 4.9 The density of a Normal distributed variable Y, where the shaded area under the density curves gives the probability that Y takes values between a and b.

In practice you can never measure an exact value y for a continuous variable, hence you always deal with an value with some amount of uncertainty, moreover you always deal with rounded values. Therefore you can only say that the value is within a small range of $[y-\varepsilon; y+\varepsilon]$. If the variable Y follows a Normal distribution N(μ ,σ^2) you can approximately compute the probability to observe a data point y in the range $[y-\varepsilon; y+\varepsilon]$ by the height of the density times 2ε which gives the area of small shaded boxes in figure 4.10: $P(y\varepsilon[y-\varepsilon; y+\varepsilon]) = f(y) \cdot 2\varepsilon$.

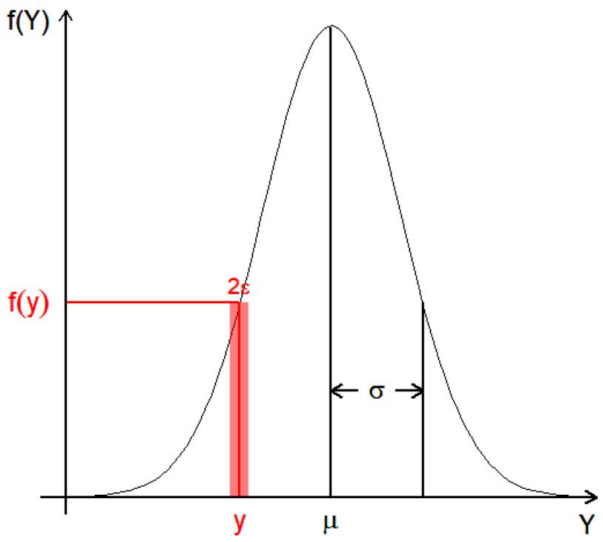

Figure 4.10 The density of a Normal distributed variable Y, where the shaded area of the box approximates the probability that Y takes values in the range $[y-\varepsilon; y+\varepsilon]$.

The probability for each y is computed by $f(y)$ times 2ε. This implies that the density $f(y)$ is proportional to the probability to observe y (see figure 4.10): $P(y\varepsilon[y-\varepsilon; y+\varepsilon]) = f(y)\cdot 2\varepsilon \sim f(y)$. This is the reason why the density $f(y)$ can be taken as likelihood for observing a (rounded) value y.

< - - - - - - Start of mayor rewrite (the sidebar has not been modified) ------- >

You can use the maximum likelihood principle to tune the two weights $w=(a,b)$ of the NN used to perform linear regression (see figure 4.6). But what is the likelihood of the observed data? In regression it is just a little bit harder to answer this question than in classification. Remember in classification you can determine the probability or likelihood for each observed value from the probability distribution with the parameters p_i. These parameters are controlled by a neural network (see figure 4.3b). In regression the observed value y_i is continuous. Here you need to work with a Normal distribution (other distributions beside the Normal are also sometime adequat and we deal with them later in chapter 5). But now we stick to the normal distribution, it has two parameters μ and σ. For start, we leave the parameter σ fixed (let's say, we set it to $\sigma=20$) and let the NN control only the parameter μ_x. The subscript x in μ_x reminds us that this parameters depends on *x*. It is determined by the network and thus μ_x depends on parameters (weights) of the network. The simple network in figure 4.6 produces a linear dependency of $\mu_x = ax+b$ on *x.* In figure 4.7 this is shown with a bold line. Older women have on average a higher blood pressure. The weights *(a,b)* themself are determined (fitted) to maximize the likelihood of the data. What is the likelihood of the data? We start with a single data point (x_i, y_i). Inspect for example the data point of the 22 year old woman who has a sbp value of 131. For that age the network predicts an average value of μ_x=111, the spread σ is fix

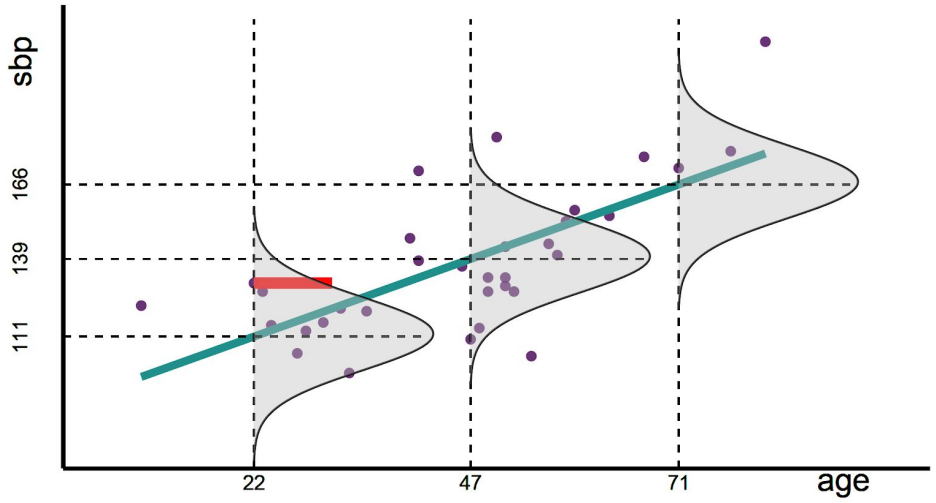

Figure 4.7 Scatterplot and regression model for the blood pressure example: The dots are the measured data points, the straight line is the linear model, the bell-shaped curves are the conditional probability distributions of the outcome Y conditioned on the observed value x. The length of the red bar is proportional to the likelihood for the observed sbp of 131 for a 22 year old woman.

In other words a woman of age 22 has (according to the model) a most likely blood pressure of 111. But other values are also possible. The probability density $f(y, \mu = 111, \sigma = 20)$ for different values of the blood pressure y is distributed around 111 value by a normal (shown in shaded grey area in figure 4.7 and in figure 4.10b).

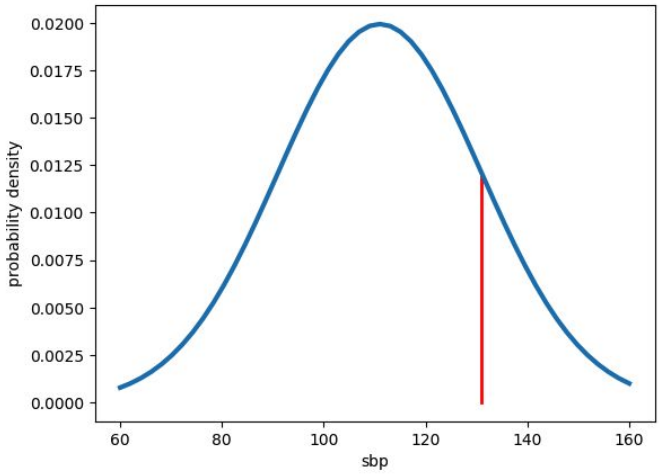

Figure 4.10b The conditional normal density function f.

Our observation has a blood pressure of 131. The likelihood of this observation is given by the density as follows:

$$f(y = 131, \mu = 111, \sigma = 20) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi \cdot 100}} e^{-\frac{(131-111)^2}{2 \cdot 400}} \approx 0.01209$$

See also the red bar in figures 4.7 and 4.10b.

For each value of the input X the output Y follows another normal distribution. For example for a woman age of 47 and sdp of 110, the parameter is μ_x =139 and the corresponding likelihood of that given blood pressure is determined by $f(y=110,\mu=139,\sigma=10)$. Because the normal probability distribution depends via $\mu_{x_i}=a.x_i+b$ on the value x_i it is often called a *conditional* probability distribution (CPD).

As before in the classification case, the likelihood of all points (we assume independence) is given by the product of the individual likelihoods, as:

$$L(a,b) = f(y_1,\mu_{x_2},\sigma)\cdot f(y_2,\mu_{x_2},\sigma)\ldots = \prod_{i=1}^{n} f(y_i,\mu_i,\sigma) = \prod_{i=1}^{n} f(y_i,a\cdot x_i+b)$$

This likelihood depends only on the parameters *a* and *b.* The values $\hat{a},\hat{b}$ maximizing it, are our best guess and are known under the name *maximum likelihood estimates*. That's also the reason the get their hat. Practically, we again take the log and minimize the negative log-likelihood.

$$l(a,b) = -log(\prod_{i=1}^{n} f(y_i,\mu_i,\sigma)) = -\sum_{i=1}^{n} log(f(y_i,a\cdot x+b))$$

Let's do this minimization in a brute force fashion by systematically varying the parameters *a* and *b* in the notebook

https://github.com/tensorchiefs/dl_book_playground/blob/master/demo_likelihood.ipynb, which main part is shown in listing 4.2.a

Listing 4.2a: Estimating the maximum likelihood solution , the brute force way

```
best_a = best_b = 0
best_like=1000000
for a in np.linspace(0,3,20): #A
    for b in np.linspace(80,200,100):#A
        mus = a*x + b #B
        neg_lok_like = -np.sum(np.log(f(y, mus))) #C
        if (neg_lok_like < best_like): #D
            best_a = a
            best_b = b
            best_like = neg_lok_like
print('Max. Lik. estimates: a ', best_a, ' b ', best_b)#E

#A Looping over all possible values of a,b
#B Estimates the parameters mu for all training data
#C Estimating the negative log-likelihood
#D Hooray we found a new minimum!
#E Max. Lik. estimates: a  1.1052...  b  87.2727...
```

In chapter 3 you have already minimized this loss function via SGD and you have received *a*=1.1 and *b*=87.8 as optimal parameter values. And indeed the maximum likelihood estimate, shown here is identical with the MSE approach of chapter 3. For the detailed derivation see the sidebar "ML based derivation of the MSE loss".

Sidebar: ML based derivation of the loss in linear regression

Let's follow step by step the maximum likelihood approach to derive the loss for a classical linear regression task. The maximum likelihood approach tells you, that you need to find those values for the weights w in the NN (here $w = (a, b)$ - see figure 4.6) that maximizes the the likelihood for the observed data. The data are provided as n pairs (x_i, y_i). The likelihood of the data is given by the following product:

$$w = argmax_w \left\{ \prod_{i=1}^{n} f(y_i|x_i, w) \right\} \Rightarrow$$

Maximizing this product leads to the same results as minimizing the sum of the corresponding negative log-likelihoods (see section 4.1).

$$w = argmin_w \left\{ \sum_{i=1}^{n} - log\, f(y_i|x_i, w) \right\} \Rightarrow$$

Now you plug in the expression for the Normal density function (see equation 4).

$$w = argmin_w \left\{ \sum_{i=1}^{n} - log\left(\frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(y_i - w_{x_i})^2}{2\sigma^2}}\right) \right\} \Rightarrow$$

Let's use the rule that $log(c \cdot d) = log(c) + log(d)$ and $log(e^g) = g$ and the fact that $(c-d)^2 = (d-c)^2$

$$w = argmin_w \left\{ \sum_{i=1}^{n} - log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{(\mu_{x_i} - y_i)^2}{2\sigma^2} \right\} \quad (5) \Rightarrow$$

Adding a constant does not change the position of the minimum and since the first term $-log(\frac{1}{\sqrt{2\pi\sigma^2}}$ is constant with respect to a and b, we can drop it $w = argmin_w \left\{ \frac{1}{2\sigma^2} \sum_{i=1}^{n} (\mu_{x_i} - y_i)^2 \right\} \Rightarrow$

Multiplying with a constant factor does not change the position of the minimum and we can freely multiply with the constant factor $2 \cdot \sigma^2/n$ so that we finally obtain the formula for the MSE loss.

$$(a, b) = argmin_w \left\{ \frac{1}{n} \sum_{i=1}^{n} (\mu_{x_i} - y_i)^2 \right\}$$

With this we have derived the loss, which we need to minimize to find the optimal values of the weights. Please note, that you only need to assume that σ^2 is constant but you do not need to derive the value of σ^2 to derive the loss function of a classical linear regression model.

$$Loss = MSE = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}i - y_i)^2 \quad (7)$$

Tata!! We ended up with the task to find the parameter values for a and be which minimizes the sum of the squared residuals. This can be done by minimizing the mean square error.

The maximum likelihood approach did indeed lead us to a loss function that is nothing else than the MSE!
In case of simple linear regression, the fitted value is $\hat{y}_i = \mu_{x_i} = a \cdot x_i + b$ yielding

$$Loss = \frac{1}{n} \sum_{i=1}^{n} (a \cdot x_i + b - y_i)^2$$

Let's take a step back and reflect what we have done so far. First, we used a neural network to determine the parameters of a probability distribution. Second, we choose a normal distribution to model our data. A normal probability distribution has two parameters μ and σ. We keep σ fix and only modelled μ_{x_i} using the simplest model possible, the linear regression $\mu_{x_i} = a * x_i + b$. The corresponding y-value (spb) to an x-value (age) is distributed like

a normal.

$$Y_{x_i} \sim N(\mu_{x_i} = a * x_i + b, \sigma^2)$$

We can extend this approach in several ways.

1. We could choose a different probability distribution beside the normal. As turns out there are certain situations, where a normal distribution is not adequat. Take for example for count data. A normal distribution always includes negative values. But some data like count data does not have negative values. We deal with those cases in chapter 5.
2. We can use a full blown neural network instead of a linear regression to model μ_{x_i}, this is done in the next section.
3. We could also model σ, and allow uncertainty to increase. This is done in section 4.3.3.

< - - - - - - End of mayor rewrite (the sidebar has not been modified) ------- >

4.3.2 Using a NN with hidden layers to model non-linear relations between input and output

A NN without hidden layer (see figure 4.6) models a linear relationship between input and output: $out = a \cdot x + b.$. Now you can extend this model and use a NN with one or more hidden layers to model μ_x. Let's still assume that the variance σ^2 is constant. With the NN in figure 4.12 you model for each input x a whole conditional probability distribution for the output which is given by:

$$Y_{x_i} \sim N(\mu_{x_i}, \sigma)$$

You will see in the following that the means of these CPDs do not need to be aligned along along a line, if you add at least one hidden layer to your NN (see figure 4.12)

In Listing 4.3 you see how you can simulate some data from a sinus function and fit a NN with 3 hidden layers and an MSE loss to the data, resulting in a well fitting non-linear curve (see figure 4.11).

Listing 4.3 A fully connected NN with 3 hidden layers to model non-linear relation ships using the MSE loss

```
n = 300  # A

np.random.seed(32)

x = np.linspace(0,1*2*np.pi,n)

y1 = 3*np.sin(x)

y1 = np.concatenate((np.zeros(60), y1+np.random.normal(0,0.15*np.abs(y1),n),np.zeros(60)))

x=np.concatenate((np.linspace(-3,0,60),np.linspace(0,3*2*np.pi,n),np.linspace(3*2*np.pi,3*2*np.pi
+3,60)))

y2 = 0.1*x+1

y=y1+y2


model = Sequential()  #B

model.add(Dense(1, activation='relu',batch_input_shape=(None, 1)))
```

```
model.add(Dense(20, activation='relu'))

model.add(Dense(50,activation='relu'))

model.add(Dense(20, activation='relu'))

model.add(Dense(1, activation='linear'))

opt = optimizers.Adam(lr=0.0001)

model.compile(loss='mean_squared_error',optimizer=opt)

history=model.fit(x, y,
                  batch_size=n,
                  epochs=10000,
                  verbose=0,
                 )   # C


# A Simulate some data from a sinus function with a small slope and non constant noise

# B define the fully connected NN with 3 hidden layers

# C fit the NN using the MSE loss
```

With this extension you can model arbitrary complicated non-linear relationships between input and output such as for example a sinus (see figure 4.12).

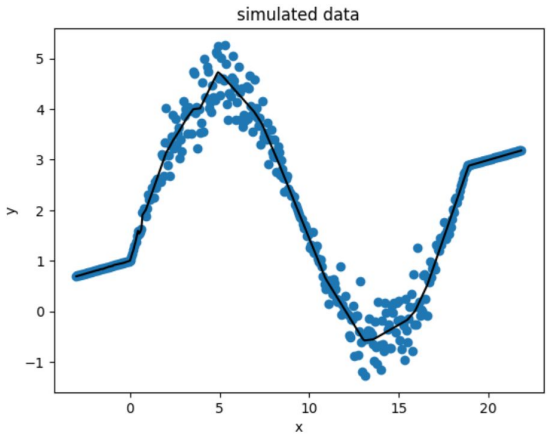

Figure 4.11 A sinus shaped curve (solid line) was fitted to the data points (see dots) by using a fully connected NN with three hidden layers and an MSE loss.

How does this work? In chapter 2 we discussed that hidden layers allow to construct new features from the input feature. For example, a NN with one hidden layer holding 8 neurons (see figure 4.12) allows the NN to construct 8 new features from the input x.

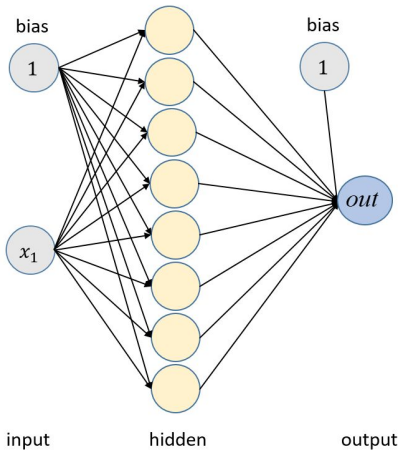

Figure 4.12 Extended linear regression where the 8 neurons in the hidden layer with linear activation function give the features from which the output "out" is computed.

Then the NN models a linear relationship between these new features and the outcome. The derivation of the loss function stays the same and leads to the MSE loss formula

$$Loss = MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}i - y_i)^2 \quad (7)$$

In the case of NN with hidden layer (see for example figure 4.12) the modeled output $\hat{y}_i = f_{NN_{4.11}}(x, w)$ is a quite complicated function of the input x_i and all weights in the NN. This is the only difference to a simple linear model that is encoded by a NN without hidden layer (see figure 4.6) where the fitted value is a simple linear function of the weights and the input $\hat{y}_i = f_{NN_{4.6}}(x, a, b) = a \cdot x_i + b$.

4.3.3 Using a NN with an additional output for regression tasks with heteroscedasticity

One assumption in classical linear regression is *homoscedasticity*, meaning that the variance of the outcome does not depend on the input value *x*. Therefore you needed only one output node (see figure 4.6 and 4.11) to compute the first parameter μ_x of the conditional Normal distribution. If you also allow the second parameter, σ_x, to depend on *x* then you need a NN with a second output node. If the variance of the output's CPD is not constant but dependent on *x*, we talk about *heteroscedasticity*.

Technically you can easily realize this by adding a second output node (see figure 4.13). Because the NN in figure 4.12 has also a hidden layer, it does allow for a non-linear relationship between the input and the outputs. The two nodes in the output layer provide the parameter values μ_x and σ_x of the CPD $N(\mu_x, \sigma_x^2)$. When working with a linear activation function in the output layer, you can get negative and positive output values. Thus, the second output is not directly taken as standard deviation σ_x, but as $log(\sigma_x)$. The standard deviation is then computed from out_2 via $\sigma_x = e^{log(\sigma_x)}$ ensuring that σ_x is a non-negative number.

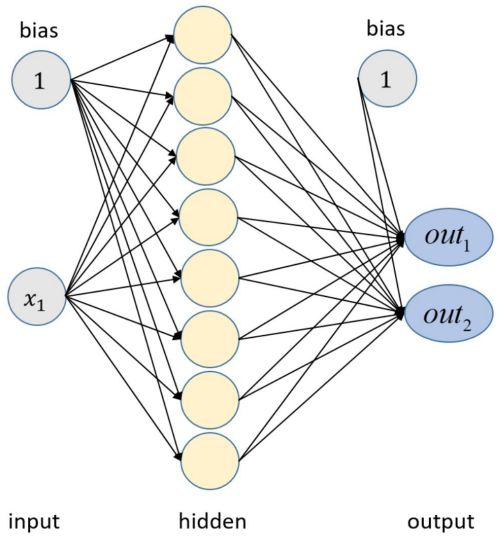

Figure 4.13 A NN with two output nodes can be used for regression tasks with heteroscedasticity

Because the classical linear regression assumes that the variance σ^2 is constant (called homoscedasticity assumption), you might suspect that it makes things much more complicated if you want to allow for varying σ^2 called heteroscedasticity. But this is not the case. The homoscedasticity assumption is only used in the derivation of the loss function to get rid of the terms containing σ^2 leading to the MSE loss. The MSE loss is provided by all DL frameworks and also by Keras and TensorFlow. But it is also possible to define custom loss functions (see listing 4.4). In the regression case with non-constant variance (heteroscedasticity) the loss is still given by negative log-likelihood defined in equation (5) but cannot be further simplified.

$$(weights) = argmin_{(weights)} \left\{ \sum_{i=1}^{n} -log\left(\frac{1}{\sqrt{2\pi\sigma_{x_i}^2}}\right) - \frac{(\mu_{x_i} - y_i)^2}{2\sigma_{x_i}^2} \right\} \quad (5)$$

With the loss

$$loss = \sum_{i=1}^{n} -log\left(\frac{1}{\sqrt{2\pi\sigma_{x_i}^2}}\right) e^{-\frac{(\mu_{x_i} - y_i)^2}{2\sigma_{x_i}^2}}$$

Optimizing this loss is not a problem at all. You can again use the SGD machinery to tune the weights for a minimal loss. In tensorflow you can realize this by defining a custom loss and then using this custom loss for the fitting procedure (see listing 4.4).

Listing 4.4 A fully connected NN with hidden layers, two output nodes and a custom loss to fit al non-linear heteroscatic regression model

```
tf.reset_default_graph() #Just to be sure to start with an empty graph


X = tf.placeholder('float32', shape=[None,1], name='x_data')

Y = tf.placeholder('float32', shape=[None,1], name='y_data')


W1 = tf.layers.dense(inputs=X,units=20,activation='relu', name ='W1') # B
```

```
W2 = tf.layers.dense(inputs=W1,units=50,activation='relu', name ='W2')

W3 = tf.layers.dense(inputs=W2,units=20,activation='relu', name ='W3')


mu = tf.layers.dense(inputs=W3,units=1,activation='linear', name ='mu')

log_sigma = tf.layers.dense(inputs=W3,units=1,activation='linear', name= 'log_sigma')


a=1/(tf.sqrt(2*3.141*tf.square(tf.exp(log_sigma)))) # A

b1=tf.square(mu-Y)

b2=2*tf.square(tf.exp(log_sigma))

b=b1/b2


loss = tf.reduce_sum(-tf.log(a)+b,axis=0) # A

train_op = tf.train.AdamOptimizer(0.00003).minimize(loss) # C

init_op = tf.global_variables_initializer()


# A Define a custom loss

# B Define a NN with 3 hidden layers and 2 outcome nodes

# C Use the custom loss for the fitting
```

You can now visualize the fit by not only plotting the curve of the fitted values but also curves for the fitted values plus/minus the fitted standard deviations to illustrate the varying spread of the fitted CPD (see figure 4.14).

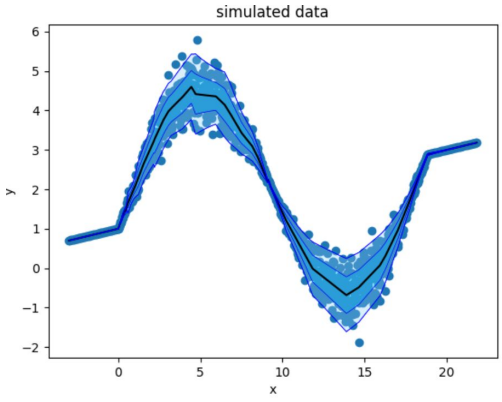

Figure 4.14 The fitted values follow a sinus shaped curve (solid middle line give the position of fitted μ_x) with varying standard deviation (the two thin outer lines correspond to $\mu \pm \sigma$ and $\mu \pm 2\sigma$). A NN with three hidden layers, two output nodes, and a customized loss was used to fit the data points (see dots) .

You can design this network arbitrary deep and wide to model complex relationships between x and y. If you only

want to allow for linear relationships between inputs and outputs you should use a NN without hidden layers.

Hands-on time: Exercise: Open the notebook https://github.com/tensorchiefs/dl_book/blob/master/chapter_04/nb_ch04_03.ipynb and work through the code until you reach the first exercise, indicated by a pen icon. You'll see how to simulate the data shown in figure 4.14. Your task in the first exercise is to fit a linear model determine the loss of this fit ??? .

You have seen now, how to derive a loss function with the maximum likelihood approach. You only need a parametric model for your data. The probability distribution of this model gives the likelihood of the data. You then design a NN that outputs the parameters of the probability distribution. The rest is done by tensorflow or keras: you use SGD to find the values of weights in the NN that leads to modeled parameter values that minimize the negative log-likelihood. In many cases the negative log-likelihood is provided as pre-defined loss function, such as the crossentropy or the MSE loss. But you can also work with arbitrary likelihoods by defining a customized loss function that corresponds to the negative log-likelihood - you have seen in listing 4.4 a customized loss defining the negative log likelihood for regression problems with non-constant variance.

4.4 Summary

- In the maximum likelihood approach you tune the parameter of a model such that the resulting model can produce the observed data with higher probability than all other models with different parameter values.
- The maximum likelihood approach is a versatile tool to fit the parameters of models. It is widely used in statistics and provides a sound theoretical framework to derive loss functions
- To use the maximum likelihood approach, you need to define a parametric probability distribution for the observed data
    - The likelihood of a discrete variable is given by a discrete probability distribution
    - The likelihood of a continuous outcome is given by a continuous density function
- The maximum likelihood approach involves:
    - Defining a parametric model for the (discrete or continuous) probability distribution of the observed data
    - Maximizing the joined likelihood for all observed data - the resulting parameter values are also called Maximum Likelihood (ML) estimators.
- Using the ML approach for a classification task is based on a Bernoulli or multinomial probability distribution and leads to the standard crossentropy loss
- Using the ML approach for a linear regression task is based on a Normal probability distribution and leads to the mean squared error (MSE) loss
- For linear regression, we can interpret the output of the network, as the parameter μ_x of a conditional Gaussian distribution, which is called conditional because the parameter μ_x depends on the input.
- Regression with non-constant variance can be fitted with a loss that correspond to the negative log-likelihood of a Normal probability model, where both parameters (mean and standard deviation) can be computed by a NN with 2 outputs.
- Non-linear relations can be fitted with the MSE loss by introducing hidden layers
- For the binary classification, we can interpret the output of the network as the parameter *p* of a logistic regression.
- For linear regression, we can interpret the output of the network, as the parameter μ of a Gaussian distribution.
- Generally, likelihoods of arbitrary probability distributions can be maximized in the framework of NN by interpreting the output of the NN as the parameters of the probability distributions.