3 Regression and optimization

We have outlined supervised learning somewhat in the introduction, and it is now time to discuss it in more depth.

3.1 Linear regression and gradient descent

Linear regression is usually taught in high school, though I hope that this course will provide a new appreciation of this method. Linear regression is the simplest form of machine learning and it is an excellent example to introduce some methods and describe some of the challenges. Supervised machine learning is essentially regression with an emphasis on high-dimensional data with nonlinear relations. While linear regression seems limited in scope, many problems are at least piecewise linear or linear on short scales so that linear methods have still some practical relevance.

size (sqft)	price (k$)
937	187
1150	222
1170	330
1290	310
1275	290
1410	440
1550	600
1730	550
1910	600

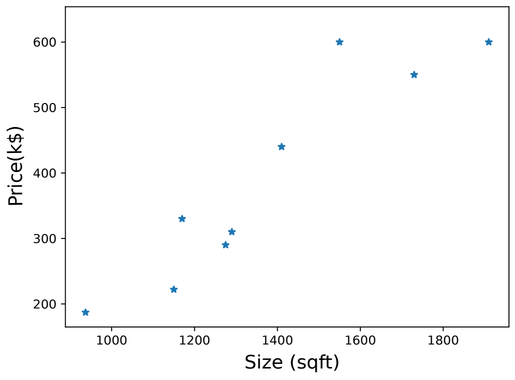

Fig. 3.1 Some example data of house prices and the corresponding size of each house

To discuss linear regression we will follow an example of describing house prices. The table on the left in Figure 3.1 lists the size in square feet and the corresponding asking prices of some houses. These data points are plotted in the graph on the right in Figure 3.1. The question is, can we predict from these data the likely asking price for houses of different size?

To do this prediction we make the assumption that the house price depend essentially on the size of the house in a linear way. That is, a house twice the size should cost twice the money. Of course, this linear model does clearly not capture all the dimensions in the data. Some houses are old while others might be new and modern. Some houses might need repair and other houses might have some special features.

Of course, as everyone in the real estate business knows it is also the location that is very important. Thus, we should keep in mind that there might be unobserved, so called **latent** dimensions in the data that might be important in explaining the relations. However, we do ignore such hidden causes at this point and just use the linear model as our hypothesis.

The linear model of the relation between the house size and the asking price can be made mathematically explicit with the linear equation

$$p(s;a,b) = as + b, \tag{3.1}$$

where p is the asking price, s is the size of the house, and a and b are model parameters. Note that p is a function of s, and we follow here a notation where the parameters of a function are included after a semicolon. If the parameters are given, then this function can be used to predict the price os a house for any size. This is the general theme of supervised learning; we make assume a specific function with parameters that we can use to predict new data.

The remaining question is now about what values these constants should have? To determine the values for the model parameters we must define how we evaluate the goodness of the values. This evaluation is specified in a loss function $\mathcal{L}$. We will use here the common choice of the mean square error (MSE) function

$$L(a,b;s^{(i)},p^{(i)}) = \frac{1}{2N}\sum_i (as^{(i)} + b - p^{(i)})^2. \tag{3.2}$$

The superscript i labels the different training examples and we put this into brackets so it is not confused with an exponent. This function considers the square distance between the predicted price values (the linear function) and the actually asking prices. Note that we view this function as a function of the model parameters a and b (the unknowns), and the training data $s^{(i)}$ and $p^{(i)}$ are the parameters of this function that are given to us. The loss function is central in machine learning and designing the right loss function is an important discussion for us in later chapters.

We use the loss function to determine the model parameters. More specifically, the values of the model parameters that minimize the average of theses distances are considered to result in the best predictor for new data points. To find these values we have to minimize the loss function as functions of the parameters. Since the loss function here is a sum of square functions, this can be calculated analytically and graduate students will be asked to calculate this. We will however use this opportunity to introduce the method called gradient descent that is a dominating technique in machine learning. The idea is to start with random value for the parameters and improve them by changing the values along the negative gradient,

$$\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} a \\ b \end{pmatrix} - \eta \nabla L(a,b;s^{(i)},p^{(i)}) \tag{3.3}$$

The **hyperparameter** η is called the learning rate, and the Nabla operator ∇ represents the gradient.

$$\nabla L(a,b;s^{(i)},p^{(i)}) = \begin{pmatrix} \frac{\partial}{\partial a} \\ \frac{\partial}{\partial b} \end{pmatrix} L(a,b;s^{(i)},p^{(i)}). \tag{3.4}$$

Minimizing the loss function with gradient descent is an important part of machine learning and the basic principle to derive the learning rule for specific models. In our case, calculating the derivative for MSE loss function (3.9) with the linear prediction function (3.1) gives the following learning rules for the two parameters

$$a \leftarrow a - \eta \frac{1}{N} \sum_i (as^{(i)} + b - p^{(i)}) s^{(i)} \tag{3.5}$$

and

$$b \leftarrow b - \eta \frac{1}{N} \sum_i (as^{(i)} + b - p^{(i)}) \tag{3.6}$$

The results of the regression for different numbers of iterations is shown in Figure 3.1. In the next chapter we discuss the implementation in detail.

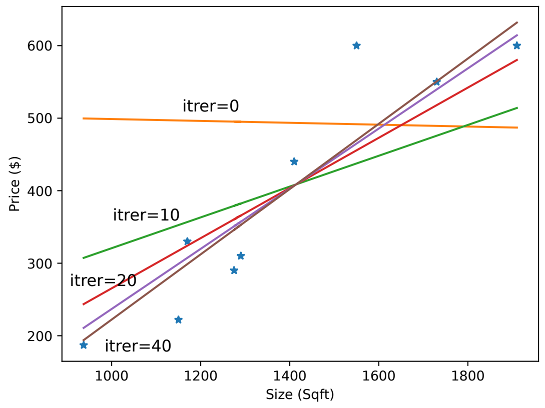

Fig. 3.2 Example of regressing dat with a linear model. The data points shown as stars specify some house prices together with the size of the house. The different lines show how a gradient descent with an appropriate choice of paramaters would result in a linear regression lines for different numbers of iterations

3.2 Error surface and challenges for gradient descent

Here is where the fun starts. Let us implement the whole procedure. The first part is easy; just link pylab and some plotting routines we need later, assign two array with the raw numbers, and plot them to see how they look like.

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
hsize=array([937,1150,1170,1290,1275,1410,1550,1730,1910]);
price=array([187,222,330,310,290,440,600,550,600]);
```

We now write the regression code. First we set the starting values for the parameters a and b, and we initialize an empty array to store the values of the loss function L in each iteration. We also set the update (learning) rate η to a small value. We then do 10 iterations to update the parameters a and b with the gradient descent rule.

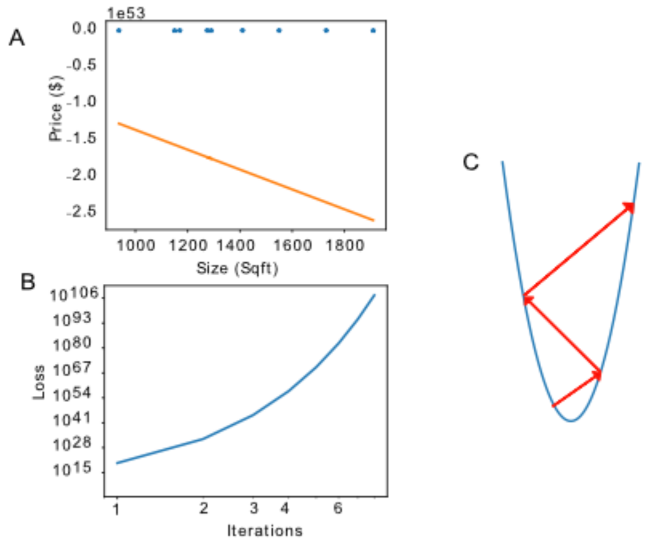

Fig. 3.3 (First attempt to implement a gradient descent learning rule for linear regression.

```
a=array([-1]); b=array([-1]); L=array([])
alpha=0.1
for iter in range(10-1):
    y=a[-1]*hsize+b[-1]
    a=append(a,a[-1]-alpha*sum((y-price)*hsize))
    b=append(b,b[-1]-alpha*sum(y-price))
    L=append(L,sum((y-price)**2))
plot(hsize,price,'*')
plot(hsize,y)
xlabel('Size␣(Sqft)')
ylabel('Price␣($)')
show()
loglog(L)
show()
```

The result of this program is shown in Fig. 3.3. The fit in on the left does not look right at all. To see what is going on it is good to plot the values of the loss function as shown in Fig. 3.3B. As can be seen, the loss function gets bigger and the values are also very large.

The rising loss value is a hint that the learning rate is too large. The reason that this can happen is illustrated in Fig. 3.3C that shows a quadratic loss surface. When the update term is too large to overshoot the minimum, then the next step can even be higher, and since the slope at this point is also higher, every step can increase the loss value.

So, let's try it again with a much smaller learning rate of `alpha=0.00000001` which was chosen after several trials to get what look like the best result. The results shown in Fig. 3.2 look certainly much better though also not quite right. The fitted curve does not seem to balance the data points well, and while the loss values decrease at first rapidly, they seem to get stuck at a small value.

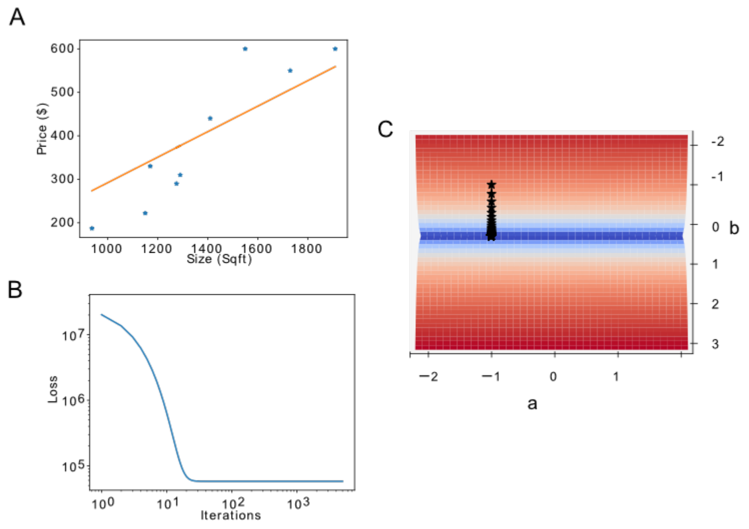

Fig. 3.4 Second attempt to implement a gradient descent learning rule for linear regression with a much smaller learning rate and more iterations.

To see more closely of what is going on we can plot the loss function for several values around our expected values of the variable. This is shown in Fig. 3.2C. This reveals that the change of the loss function with respect to the parameter b is large but that changing the parameter a on the same scale has little influence on the loss value. To fix this problem we would have to change the learning rate for each parameter, which is not practical in higher dimensional models. There are much more sophisticated solutions such as Amati's Natural Gradient, but a quick fix for many applications is to normalize the data so that there ranges is between zero and one. Thus, by adding the code

```
hsize=(hsize-min(hsize))/(max(hsize)-min(hsize))
price=(price-min(price))/(max(price)-min(price))
```

and setting the learning rate to `alpha=0.04` we get the solution shown in Fig. 3.2 The solution is much better, although the learning path is still not optimal. However, this is a solution that is sufficient most of the time.

3.3 Regularization: Ridge regression and LASSO

We discussed above a linear problem with a one dimensional input x. Machine learning problems often consist of high dimensional problems in the sense that the input is a

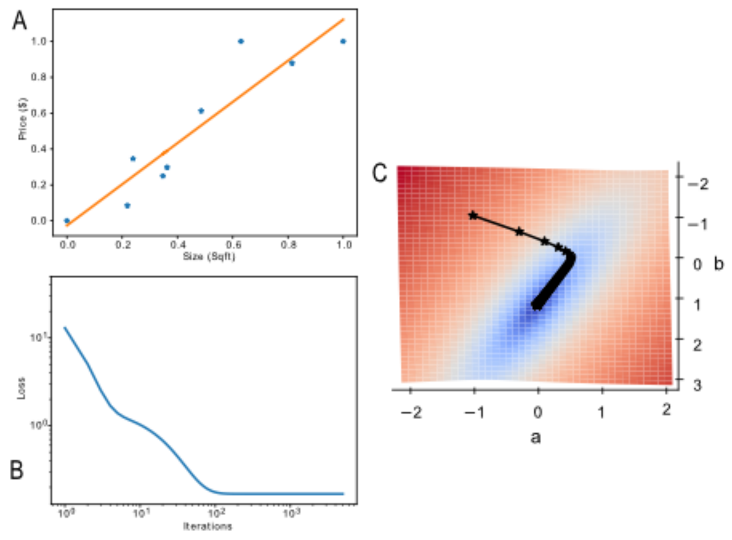

Fig. 3.5 Third attempt which leads to a much better solution by simply normalizing the range of the data to a unitary regime. .

vector with many dimensions. For example, if we want to do image processing, we would represent a grey image as a list of many grey level values, one for each pixel. Within linear regression this means that we should introduce one parameter for each input dimension. Such a linear model would now look like

$$y = w_0 + w_1x_1 + w_2x_2 + ... = \mathbf{w}^{\mathrm{T}}\mathbf{x} \tag{3.7}$$

This shows how useful vector notation are to summarize all the components. Also, we made a common trick to augment the feature vector with a constant $x_0 = 1$ so that we do not have to treat the y-intercept differently to the other parameters.

It is very common in machine learning applications that the list of input feature is very large. A good example is image processing where the input are the pixels of an image, which can run into the millions of values. Also, it has been common in the past to enlarge the input vector by supplying combinations of input features and higher order moments of the features to help in modelling nonlinear relations. Hence, the number of parameters is commonly large while it is not certain how important the input features are. We have seen that a large amount of parameters relative to the number of data can lead to overfitting, and methods to restrict the parameters to prevent overfitting has been a very important part of machine learning. A common consideration is to include a term in the objective function that keeps the parameters small. For example, we can include a penalty term proportional to the absolute value of the parameters or even the sum of the square values to even more penalize larger weight values. A penalty on the size of the parameter value should keep these values small or even zero in case they do not contribute sufficiently to the model. To be concrete, if we define the L^p norm as

$$||\mathbf{w}||_p = \left(\sum_i |w_i|^p\right)^{\frac{1}{p}}, \tag{3.8}$$

then we can modify the previous MSE objective function as

$$L(\mathbf{w};\mathbf{x}^{(i)},y^{(i)}) = \frac{1}{2N}\sum_i(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} - y^{(i)})^2 + \eta||\mathbf{w_j}||_p^p. \tag{3.9}$$

The parameter η does thereby set the scale of the penalty. The two most use methods use the L^1 norm, which is the basis for the method called **LASSO** (Least Absolute Shrinkage and Selection Operator), and the L^2 norm in the method called **Ridge regression**. In the assignments we will implement these methods directly, and we will just use the routines from the sklearn toolbox to look at an instructive example. In this example we chose points from the linear model

$$y = 0.5x_1 + 0.5x_2 \tag{3.10}$$

and and some noise in the feature values. Our training points are hence $(x_1, x_2, y) = \{(0,0,0),(1,0.9,1),(2,2.1,2)\}$. The listing is shown below and the results are plottet in Fig. 3. The results look very similar, but it is instructive to look at the coefficients, which are

$$\{w_1, w_2\}_{\text{reg}} = \{1.82, -0.91\} \tag{3.11}$$
$$\{w_1, w_2\}_{\text{ridge}} = \{0.45, 0.42\} \tag{3.12}$$
$$\{w_1, w_2\}_{\text{lasso}} = \{0.82, 0\} \tag{3.13}$$

The values of the linear regression without normalization are interesting as the large value for the first parameter is compensated by the large negative value of the second parameter. While this works for the specific training points, generalization with other examples might be not as good. Ridge regression finds much closer values similar to our original model. However, LASSO is also interesting as it finds a much smaller model where it can explain the data with only one parameter. This might be valuable in practical applications.

```
from pylab import *
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(); ax = fig.gca(projection='3d')

x = array([[0, 0], [1, .9], [2.1, 2]]); y= array([0, 1, 2])
ax.plot(x[:,0],x[:,1],y,'*');

reg = linear_model.LinearRegression(); reg.fit(x, y)
print(reg.coef_)
ax.plot([0,2],[0,2],[0,dot(reg.coef_,[2,2])]);

reg = linear_model.Ridge(alpha = .5); reg.fit(x, y)
print(reg.coef_)
```

```
ax.plot([0,2],[0,2],[0,dot(reg.coef_,[2,2])]);

reg = linear_model.Lasso(alpha = 0.1); reg.fit(x, y)
print(reg.coef_)
ax.plot([0,2],[0,2],[0,dot(reg.coef_,[2,2])]);

show()
```

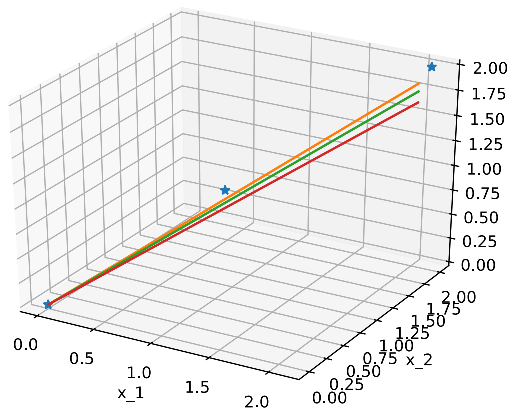

Fig. 3.6 (linear regression without regularization and with regularization using Ridge regression and Lasso.

Regression is an important topic in machine learning, and there are other methods that are actively used. For example, a method called **Dropout** is an algorithm that includes a certain probability that some of the parameters are set to zero during the generation of an output during training. This is particularly used in deep neural networks where it is thought to resembles some biological processes as neurons sometimes do not fire even with the same stimulus that would activate them at other times. In this way it is necessary that the model is able to represent specific data points with a combination of other parameters. This prevents that specific parts of the program specialize to specific sample data.

3.4 Nonlinear regression

Linear regression is simple and often applied as many relations are approximately linear, at least on small scales. However, many more challenging applications have nonlinear relations, and capturing such relations is mostly our aim in the following. To start discussing such cases let us first consider a polynomial of order n

$$y = w_0 + w_1 x^1 + w_2 x^2 + ... + w_n x^n = \sum_{i=0}^{n} w_i x^i. \tag{3.14}$$

We can fit this function to our house data with a gradient descent rule noting that the derivative (we only have one dimension at the moment) is

$$\frac{\mathrm{d}y}{\mathrm{d}w_i} = x^i. \tag{3.15}$$

Hence the learning rule is

$$w_i \leftarrow w_i - \eta x^i. \tag{3.16}$$

A fit of the house data with a polynomial of order 12 is shown in Fig. 3.4A and the corresponding Loss values at the end of training in Fig. 3.4B. The loss becomes a little bit better over time, and it might be possible to reduce this further with some more training iterations. The resulting regression curve seems to capture some of the curvature of the data, although it seems unreasonable to assume that larger houses become cheaper after a certain size.

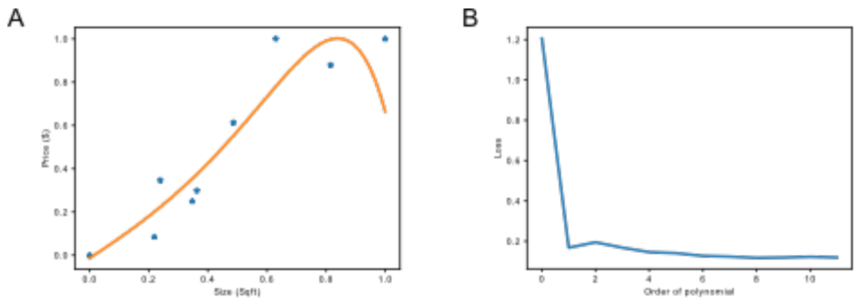

Fig. 3.7 (Polynomial regression of the house data .

While the above polynomial model function is certainly non-linear, it is interesting to note that we can cast the regression problem again into a linear framework. That is, we can create a new feature vector $\mathbf{x}$ with components $x_i = x^i$. We can then rewrite the model as

$$y = \mathbf{w}^{\mathrm{T}}\mathbf{x}, \tag{3.17}$$

which is again a linear model. In other words, we have first applied a nonlinear transformation from the feature space of $\mathbf{x}$ into a new feature space of $\mathbf{x}_{\mathrm{new}}$,

$$\mathbf{x}_{\mathrm{new}} = \phi(\mathbf{x}). \tag{3.18}$$

After this transformation we were able to use a linear model in this transformed space. The transformed space was here larger than the original space, but in this transformed space we were able to use a linear regression model. This is sometimes called generalized linear regression. Finding the right transformation function ϕ is not easy and we somewhat transformed the nonlinear problem into the problem of finding the appropriate nonlinear transformation. We will see later that this way of thinking translates well into understanding some advanced machine learning approaches.

It is worthwhile to consider another nonlinear example function to our house data. For this we choose a series of Gaussian functions

$$y = \sum_i a_i e^{-(b_i - x)^2/c_i}. \tag{3.19}$$

We have three sets of parameters in this model, and the corresponding learning rules are

$$a_i \leftarrow a_i \tag{3.20}$$
$$b_i \leftarrow b_i \tag{3.21}$$
$$c_i \leftarrow c_i \tag{3.22}$$

A fit of the data with a single Gaussian is shown in Fig. 3.4A which looks somewhat similar to the fit with the twelves order polynomial, albeit with only three parameters. However, the reason to bring up this function is to realize that with an increasing number of parameters and hence complexity of the function we can easily overfit. For example, if we take the number of Gaussians to be equal to the number of training points, we can produce a curve that goes practically through al the training points as shown in and we choose a Fig. 3.4B. It is clear that this function would purely generalize as we do not expect that houses outside these training examples cost nothing as predicted by the model. This is a clear demonstration of overfitting as mentioned in the first chapter.

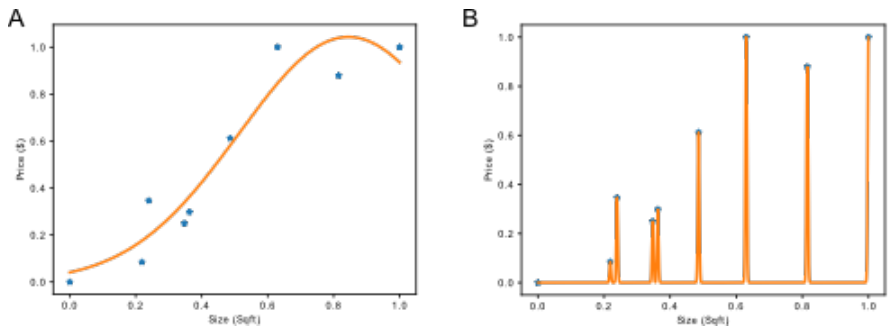

Fig. 3.8 (Nonlinear regression with a sum of Gaussians.

3.5 Advanced optimization (learning)

Learning in machine learning means to find parameters of the model **w** that minimize the loss function. There are many methods to minimize a function, and each one would constitute a learning algorithm. However, the workhorse in machine learning is the **gradient descent** algorithm that we already used above. Formally, the basic gradient descent minimizes the sum of the loss values over all training examples, which is sometimes also called a **batch algorithm** as all training examples build the batch for minimization. Let us assume we have m training data, then gradient descent iterates the equation

$$w_i \leftarrow w_i + \Delta w_i \tag{3.23}$$

with

$$\Delta w_i = -\frac{\eta}{m} \sum_{k=1}^{m} \frac{\partial \mathcal{L}(y^{(i)}, \mathbf{x}^{(i)}|\mathbf{w})}{\partial w_i}. \tag{3.24}$$

We can also write this as for all parameters using vector notation and the nabla operator

$$\nabla = \tag{3.25}$$

as

$$\Delta \mathbf{w} = -\frac{\eta}{m} \sum_{i=1}^{m} \nabla \mathcal{L}^{(i)} \tag{3.26}$$

with

$$\mathcal{L}(y^{(i)}, \mathbf{x}^{(i)}|\mathbf{w}) \tag{3.27}$$

With a sufficiently small learning rate η, this will result in a strictly monotonically decreasing learning curve. The problem in the age of big data is, however, that a large number of training examples have to be kept in memory. Such batch learning seem also unrealistic biologically or in situations where training examples are only coming in over times. So called **online** algorithms that use the training data when they come are therefore often desirable. The online gradient descent would consider only one training example at a times

$$\Delta \mathbf{w} = -\eta \nabla \mathcal{L}^{(i)}, \tag{3.28}$$

and then use another training example for another update. The training examples provide a radon walk around the true gradient descent, and this algorithms is hence also called the **stochastic gradient descent (SGD**. It can be see as an approximation of the basic gradient descent algorithm. In practice it is now common to use something in between, using so called mini-batches of the training data to iterate through them. This is formally still a stochastic gradient descent, but it combines the advantages of a batch algorithm with the reality of limited memory capacities.

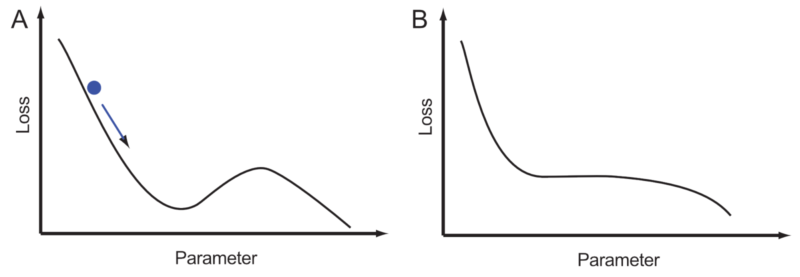

Fig. 3.9 Illustration of gradient descent with a local minima (A) and a saddle point (B).

Gradient descent is known as a very efficient local optimizer. It can often be observed that such an algorithm leads to a steep decline of the loss values before learning seems to slow down. One problem with the algorithm is that it can, strictly speaking, only find local minima as illustrated in Fig./ 3.9A. Think about a ball rolling downhill on the loss surface. With the basic gradient descent we are always strictly going downhill. However, with a real ball we would have a **momentum** so that the ball could overcome a small hill if the momentum is large enough. To incorporate momentum into the gradient descent algorithm, we can modify the update so that we take some percentage of the previous step into account,

$$\Delta \mathbf{w} = -\eta \nabla \mathcal{L}^{(i)} + \alpha \mathbf{w}, \tag{3.29}$$

A momentum term of 90% is a common starting value, but such hyperparameters of the algorithms are of course problem dependent and need to be evaluated on a case by case basis.

Local minima have often be stated as a main problem for gradient descent, although true local minima are increasingly difficult to realize in higher dimension. To be a true local minima it is necessary that all changes and all combination of changes of all directions of the parameters space lead to larger loss values. It is clear that with increasing dimensions there is therefore a increasing chance to find an "escape route". However, even so it is now realized that true local minima are likely not the problem with most high-dimensional learning scenarios, saddle points or at least shallow areas of the loss functions seems to be nevertheless a problem in many applications. A momentum term is a common way to help with such shallow areas, and adaptive learning rates and higher order methods are additional techniques for advanced supervised learning methods.

One of the most commonly used algorithms is now the ADAM optimizer. ADAM stands for Adaptive Moment Estimation which is a slight modification of the momentum method. Instead of strictly using the last entry of the gradient as the momentum, the ADAM method uses a sliding average of the gradient

$$\mathbf{m} \leftarrow \alpha_1 \mathbf{m} + (1 - \alpha_1) \nabla \mathcal{L}^{(i)}, \tag{3.30}$$

as well as the second moment of the gradient

$$\mathbf{v} \leftarrow \alpha_2 \mathbf{v} + (1 - \alpha_2) (\nabla \mathcal{L}^{(i)})^2 \tag{3.31}$$

to capture somewhat a variance of the gradient. The model parameters are then updated according to

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\mathbf{m}/(1-\alpha_1)}{\sqrt{\mathbf{v}/(1-\alpha_2)} + \epsilon}, \tag{3.32}$$

where the small factor ϵ is added to prevent possible divisions by zero.

There are many more advanced variations of gradient descent methods. For example, we could start taking into account higher order gradient terms that describe the curvature of the loss functions. While this requires the calculation of the higher derivatives or in general the Hessian, it will allow much larger learning rates that will speed up learning. One of the best variations of these methods is the **Natural Gradient** algorithm that thrives to keep the improvements in the loss function constant.

While we are not discussing these methods here in more detail, I would like to close this section by pointing out that gradient methods are of course by far not the only minimization method that can be applied to machine learning. As an example, let us consider a simple version of a **genetic algorithm**. For this we treat the vector of all model parameters as the "genom" of an individual model, and we consider a pool of such models. Each candidate model is then evaluated by the loss function, and a certain percentage of the best performers are copied into the new pool of individuals in the next generation. These parent individuals are also allowed to reproduce by taking two of these individuals and swapping parts of there genes at a certain transition point, and by changing randomly some of the entries. The first operation is called a **crossover**, while the second operation is commonly called a **mutation**. In this way we produce new candidate models that can then be again with the loss function. We will explore such an algorithm int he assignment.