Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 2

October 11th, 2022

1.3 Gradient descent

- *gradient descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- the general idea of gradient descent is to update the parameters iteratively in order to minimize the loss function
- suppose we are lost in the mountains in a dense fog, and we can only feel the slope of the ground below our feet; a good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the *steepest slope*
- this is exactly what gradient descent does: it measures the local gradient of the loss function with respect to the parameter vector $\boldsymbol{w}$, and it goes in the direction of the steepest descent, which is given by the *negative gradient vector*, because the gradient points uphill
- once the gradient is zero, a minimum of the loss function has been reached

1.3 Gradient descent

- the steps of the algorithm are the following:
  - initialize the values of the model weights, typically at random (this is called *random initialization*).
  - iteratively update the parameters one small step at a time, in the direction of the negative gradient, until the algorithm *converges* to a minimum (see Figure 2). This is done by multiplying the gradient by a predetermined positive value η (called *learning rate*) and subtracting the resulting term from the current parameter values.

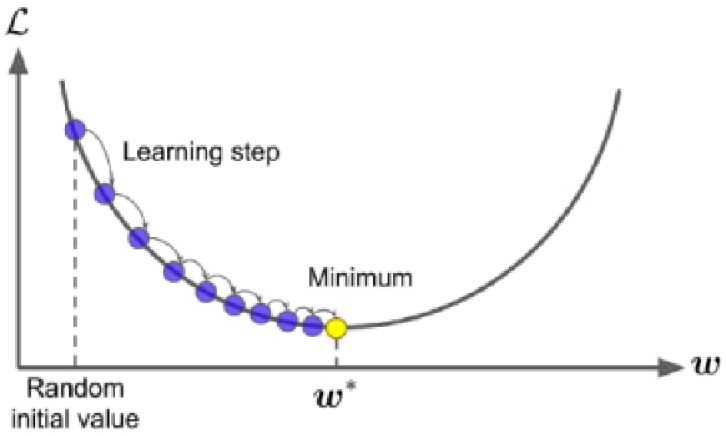


Figure 2: Gradient descent.

- we can express the update mathematically as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}),$$

  or, equivalently, as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \left(\frac{1}{n} \sum_{i=1}^{n} l^{(i)}(\boldsymbol{w}) \right),$$

  where we used (6)
- according to (7), we can compute the gradient as follows:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \frac{1}{n}(\boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^\top \boldsymbol{y}) = \frac{1}{n}\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}). \tag{8}$$

- notice that this formula involves calculations over the full training set $\boldsymbol{X}$, at each gradient descent step
- this is why the algorithm is called *batch gradient descent*: it uses the whole batch of training data at every step
- as a result it is very slow on very large training sets (but we will see much faster gradient descent algorithms shortly)
- however, gradient descent scales well with the number of features; training a linear regression model when there are hundreds of thousands of features is *much faster* using gradient descent than using the normal equation or SVD decomposition

- an important parameter in gradient descent is the size of the steps, determined by the learning rate η
- the value of the learning rate is manually pre-specified and not typically learned through model training
- this type of parameters that are tunable but not updated in the training loop are called *hyperparameters*
- *hyperparameter tuning* is the process by which hyperparameters are chosen, and typically requires that we adjust them based on the results of the training loop as assessed on a separate *validation dataset* (or *validation set*)

- if the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 3)

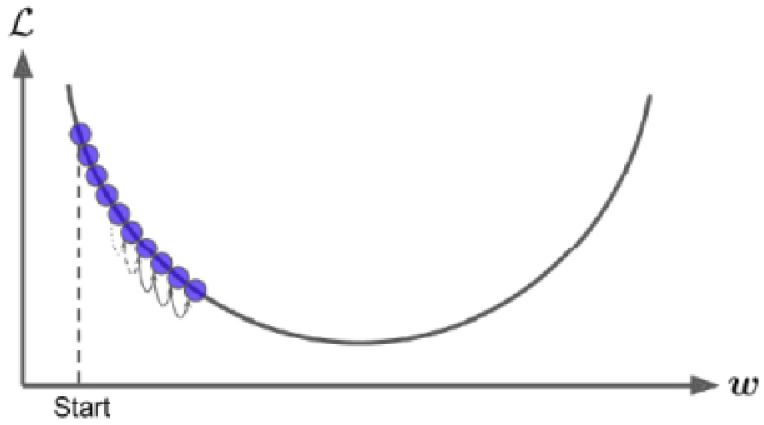


Figure 3: The learning rate is too small.

- on the other hand, if the learning rate is too high, we might jump across the valley and end up on the other side, possibly even higher up than we were before
- this might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 4)

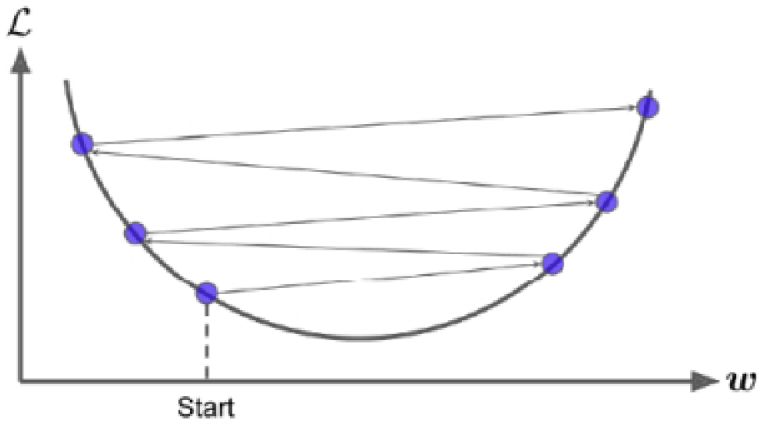


Figure 4: The learning rate is too big.

- finally, not all loss functions look like nice, regular bowls
- there may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult
- Figure 5 shows the two main challenges with gradient descent
- if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*
- if it starts on the right, then it will take a very long time to cross the plateau
- and if we stop too early, we will never reach the global minimum

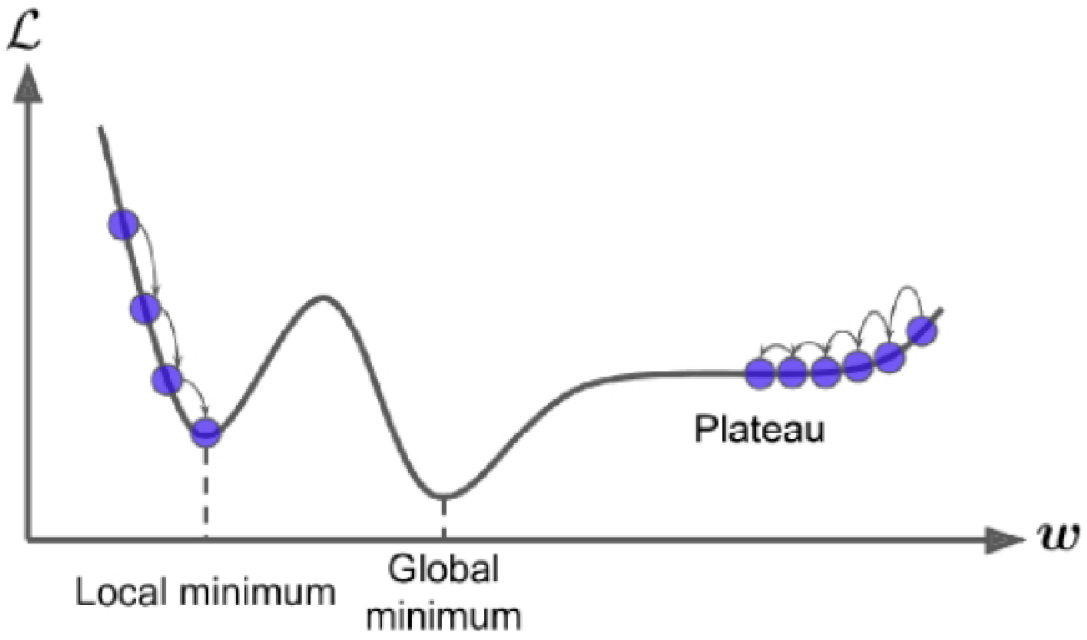


Figure 5: Gradient descent challenges.

- fortunately, the mean squared error loss function for a linear regression model happens to be a *convex function*, which means that if we pick any two points on the curve, the line segment joining them never crosses the curve
- this implies that there are no local minima, just one global minimum
- it is also a continuous function with a slope that never changes abruptly
- these two facts have a great consequence: gradient descent is guaranteed to approach arbitrarily close the global minimum (if we wait long enough and if the learning rate is not too high)

1.3 Gradient descent

- in fact, the loss function has the shape of a bowl, but it can be an elongated bowl if the features have very *different scales*
- Figure 6 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right)

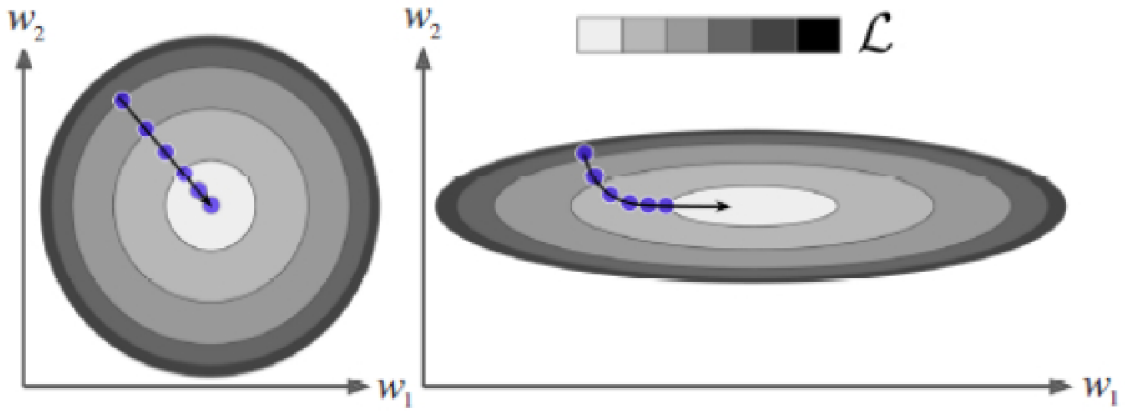


Figure 6: Gradient descent with (left) and without (right) feature scaling.

- as we can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley
- it will eventually reach the minimum, but it will take a long time
- when using gradient descent, we should ensure that all features have a *similar scale*, or else it will take much longer to converge

- the main problem with batch gradient descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large
- at the opposite extreme, *stochastic gradient descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance
- obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration
- it also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration
- mathematically, this can be expressed as:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} l^{(i)}(\boldsymbol{w}),$$

for some $1 \leq i \leq n$

- on the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the loss function will *bounce* up and down, decreasing only on average
- over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure 7)
- so once the algorithm stops, the final parameter values are good, but not optimal

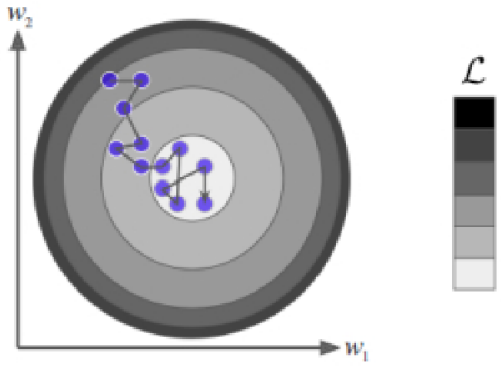


Figure 7: With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

- when the loss function is very irregular (as in Figure 5), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does
- therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum
- one solution to this dilemma is to gradually reduce the learning rate
- the steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum
- the function that determines the learning rate at each iteration is called the *learning schedule*
- if the learning rate is reduced too quickly, we may get stuck in a local minimum, or even end up frozen halfway to the minimum
- if the learning rate is reduced too slowly, we may jump around the minimum for a long time and end up with a suboptimal solution if we halt training too early

- by convention we iterate by rounds of n iterations; each round is called an *epoch*
- thus, in one epoch, each example will be used to update the weights of the model
- if we want to be sure that the algorithm goes through every instance at each epoch, we must shuffle the training set
- the last gradient descent algorithm we will look at is called *mini-batch gradient descent*
- it is simple to understand once we know batch and stochastic gradient descent: at each step, instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), mini-batch GD computes the gradients on small random sets of instances called *mini-batches*

- the algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches
- as a result, mini-batch GD will end up walking around a bit closer to the minimum than stochastic GD – but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike linear regression)
- Figure 8 shows the paths taken by the three gradient descent algorithms in parameter space during training
- they all end up near the minimum, but batch GD's path actually stops at the minimum, while both stochastic GD and mini-batch GD continue to walk around
- however, we must not forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if we used a good learning schedule

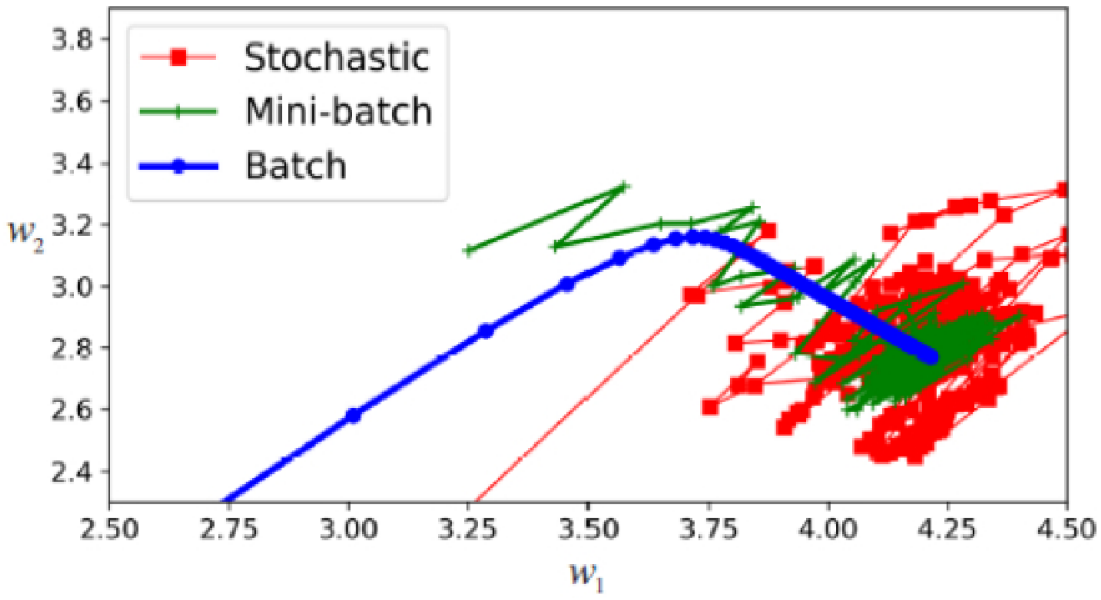


Figure 8: With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

- in each iteration, we first randomly sample a mini-batch $\mathcal{B}$ consisting of a fixed number of training examples
- we then compute the derivative (gradient) of the average loss on the mini-batch with regard to the model parameters
- finally, we multiply the gradient by a predetermined positive value and subtract the resulting term from the current parameter values
- we can express the update mathematically as follows:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \left(\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l^{(i)}(\boldsymbol{w}) \right).$$

- the set cardinality $|\mathcal{B}|$ represents the number of examples in each mini-batch (the *batch size*); the batch size is a hyperparameter of the model

- after training for some predetermined number of iterations (or until some other stopping criteria are met), we record the *estimated model weights*, denoted $\hat{\boldsymbol{w}}$
- note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards the minimizers, it cannot achieve it *exactly* in a finite number of steps
- linear regression happens to be a learning problem where there is only one minimum over the entire domain

- however, for more complicated models, like deep networks, the loss surfaces contain many minima
- fortunately, for reasons that are not yet fully understood, deep learning practitioners rarely struggle to find parameters that minimize the loss *on training sets*
- the more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*
- given the learned linear regression model $\hat{\boldsymbol{w}}^\top \boldsymbol{x}$, we can now estimate target of a new sample (not contained in the training data) given its input features
- estimating targets given features is commonly called *prediction* or *inference*

1.4 Polynomial regression

- what if our data is more complex than a straight line?
- surprisingly, we can use a linear model to fit *nonlinear data*
- a simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features – this technique is called *polynomial regression*
- so far, we considered the linear regression setting described in (2), which allowed us to fit straight lines to data using maximum likelihood estimation
- however, straight lines are not sufficiently expressive when it comes to fitting more interesting data
- fortunately, linear regression offers us a way to fit nonlinear functions within the linear regression framework: since "linear regression" only refers to "linear in the parameters", we can perform an arbitrary nonlinear transformation $\phi(\boldsymbol{x})$ of the inputs $\boldsymbol{x}$ and then linearly combine the components of this transformation

- the corresponding linear regression model is:

$$\hat{y} = \boldsymbol{w}^\top \phi(\boldsymbol{x}), \tag{9}$$

  where $\phi : \mathbb{R}^d \to \mathbb{R}^p$ is a (nonlinear) transformation of the inputs $\boldsymbol{x}$ and $\phi_k : \mathbb{R}^d \to \mathbb{R}$ is the kth component of the feature vector ϕ
- note that the feature vector model parameters still appear only linearly
- note that when there are multiple features, polynomial regression is capable of finding relationships between features (which is something a plain linear regression model cannot do)
- this is made possible by the fact that we can also add all combinations of features up to the given degree
- for example, if there were two features a and b, we would not only add the features a^2, a^3, b^2, and b^3, but also the combinations ab, a^2b, and ab^2

1.4 Polynomial regression

- consider the dataset in Figure 9; the dataset consists of $n = 10$ pairs $(x^{(i)}, y^{(i)})$, where $x^{(i)} \sim \mathcal{U}[-5, 5]$ and

$$y^{(i)} = \sin\left(\frac{x^{(i)}}{5}\right) + \cos(x^{(i)}) + \epsilon,$$

where $\epsilon \sim \mathcal{N}(0, 0.2^2)$

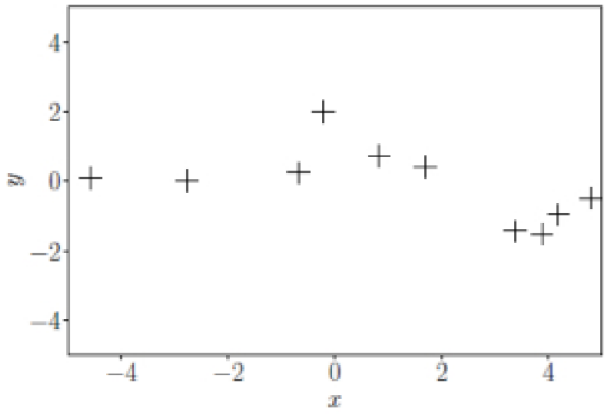


Figure 9: Polynomial regression: dataset consisting of $(x^{(i)}, y^{(i)})$ pairs, $i = 1, \ldots, 10$.

1.4 Polynomial regression

- assume that we use a polynomial model

$$\hat{y} = \boldsymbol{w}^\top \phi(x),$$

to fit it, where $\boldsymbol{w} \in \mathbb{R}^M$,

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^{M-1} \end{bmatrix} \in \mathbb{R}^M.$$

- we can evaluate the quality of a model by computing the loss $\mathcal{L}$
- instead of using this squared loss, we often use the *root mean square error (RMSE)*:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{w}^\top \phi(\boldsymbol{x}^{(i)}) - y^{(i)})^2},$$

which

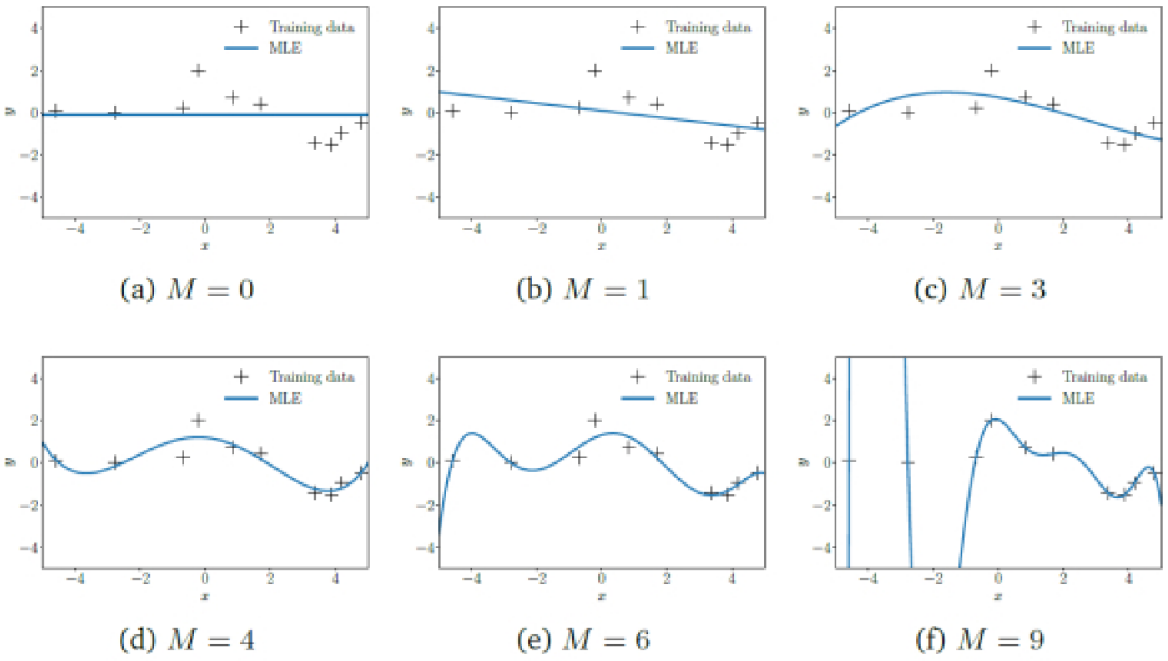
- allows us to compare errors of datasets with different sizes
- has the same scale and the same units as the observed function values $y^{(i)}$

1.4 Polynomial regression

- in order to select the best model to fit our dataset, we can use the RMSE to determine the best degree of the polynomial by finding the polynomial degree M that minimizes the RMSE
- given that the polynomial degree is a natural number, we can perform a brute-force search and enumerate all (reasonable) values of M
- for a training set of size n it is sufficient to test $0 \leq M \leq n - 1$
- for $M < n$, the maximum likelihood estimator is unique
- for $M \geq n$, we have more parameters than data points, and would need to solve an underdetermined system of linear equations, so that there are infinitely many possible maximum likelihood estimators

1.4 Polynomial regression

- Figure 10 shows a number of polynomial fits determined by polynomial regression for the dataset from Figure 9 with $n = 10$ observations
- we notice that polynomials of low degree (e.g., constants ($M = 0$) or linear ($M = 1$)) fit the data poorly and, hence, are poor representations of the true underlying function
- for degrees $M = 3, \ldots, 6$, the fits look plausible and smoothly interpolate the data
- when we go to higher-degree polynomials, we notice that they fit the data better and better
- in the extreme case of $M = n - 1 = 9$, the function will pass through every single data point
- however, these high-degree polynomials oscillate wildly and are a poor representation of the underlying function that generated the data, such that we suffer from *overfitting*



(a) $M = 0$ (b) $M = 1$ (c) $M = 3$

(d) $M = 4$ (e) $M = 6$ (f) $M = 9$

Figure 10: Maximum likelihood fits for different polynomial degrees M.

1.4 Polynomial regression

- however, the goal is to achieve good *generalization* by making accurate predictions for new (unseen) data
- we obtain some quantitative insight into the dependence of the generalization performance on the polynomial of degree M by considering a separate test set comprising 200 data points generated using exactly the same procedure used to generate the training set
- as test inputs, we choose a linear grid of 200 points in the interval of $[-5, 5]$
- for each choice of M, we evaluate the RMSE for both the training data and the test data

1.4 Polynomial regression

- looking now at the *test error*, which is a qualitative measure of the generalization properties of the corresponding polynomial, we notice that initially the test error decreases; see Figure 11 (orange)
- for fourth-order polynomials, the test error is relatively low and stays relatively constant up to degree 5
- however, from degree 6 onward, the test error increases significantly, and high-order polynomials have very bad generalization properties
- in this particular example, this also is evident from the corresponding maximum likelihood fits in Figure 10

1.4 Polynomial regression

- note that the *training error* (blue curve in Figure 11) never increases when the degree of the polynomial increases
- in our example, the best generalization (the point of the smallest *test error*) is obtained for a polynomial of degree $M = 4$

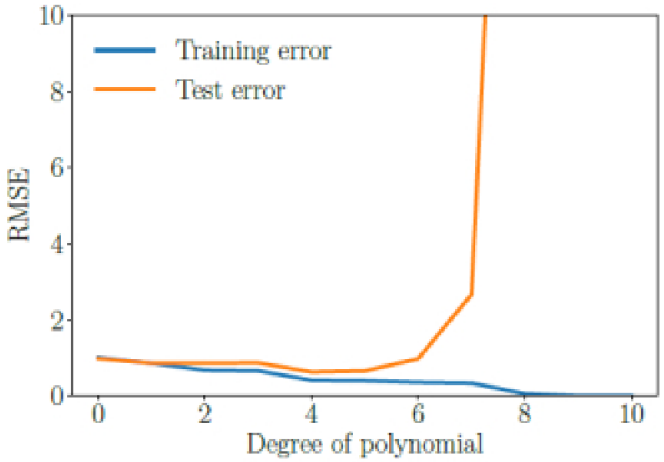


Figure 11: Training and test error.

1.4 Polynomial regression

- an important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:
  1. *Bias*. This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.
  2. *Variance*. This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.
  3. *Irreducible error*. This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).
- increasing a model's complexity will typically increase its variance and reduce its bias
- conversely, reducing a model's complexity increases its bias and reduces its variance
- this is why it is called the *bias/variance trade-off*

- we just saw that maximum likelihood estimation is prone to overfitting
- we often observe that the magnitude of the parameter values becomes *relatively large* if we run into overfitting
- to mitigate the effect of huge parameter values, we can place a *prior distribution* $p(\boldsymbol{w})$ on the parameters
- the prior distribution explicitly encodes what parameter values are plausible (before having seen any data)
- for example, a Gaussian prior $w \sim \mathcal{N}(0, 1)$ on a single parameter encodes that parameter values are expected to lie in the interval $[-2, 2]$ (two standard deviations around the mean value)

- once a dataset $\boldsymbol{X}, \boldsymbol{y}$ is available, instead of maximizing the likelihood, we seek parameters that maximize the posterior distribution $p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})$
- this procedure is called *maximum a posteriori* estimation
- the posterior over the weights $\boldsymbol{w}$, given the training data $\boldsymbol{X}, \boldsymbol{y}$, is obtained by applying Bayes' theorem as:

$$p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{y}|\boldsymbol{X})}.$$

- to find the maximum a posteriori estimation, we minimize the *negative log-posterior distribution* with respect to $\boldsymbol{w}$:

$$-\log p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = -\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) - \log p(\boldsymbol{w}) + \log p(\boldsymbol{y}|\boldsymbol{X}). \tag{10}$$

- we see that the log-posterior in (10) is the sum of the log-likelihood $\log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$ and the log-prior $\log p(\boldsymbol{w})$, so that the maximum a posteriori estimate will be a "compromise" between the prior (our suggestion for plausible parameter values before observing data) and the data-dependent likelihood

- first, assume that the weights are normally distributed with variance τ, $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \tau^2 \boldsymbol{I})$, which implies that:

$$-\log p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = \frac{1}{2\sigma^2}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2\tau^2}\boldsymbol{w}^\top \boldsymbol{w} + \text{const.} \tag{11}$$

  where const. represent terms that are independent of $\boldsymbol{w}$
- here, the first term corresponds to the contribution from the log-likelihood, and the second term originates from the log-prior
- by taking the gradient of (11) and making it equal with $\boldsymbol{0}^\top$, the maximum a posteriori estimate for $\boldsymbol{w}$ will be:

$$\boldsymbol{w}^* = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y},$$

  where we denoted $\lambda := \frac{\sigma^2}{\tau^2}$

1.5 Regularized linear models

- similarly as for maximum likelihood estimation, we can define a loss corresponding to maximum a posteriori estimation:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\lambda||\boldsymbol{w}||_2^2,$$

  where $||\boldsymbol{w}||_2 = \left(\sum_{j=0}^{d} |w_j|^2\right)^{1/2}$ is the ℓ_2 norm of $\boldsymbol{w}$
- linear regression using this loss function is called *ridge regression*
- ridge regression is a *regularized* version of linear regression: a regularization term equal to $\frac{1}{2}\lambda||\boldsymbol{w}||_2^2$ is added to the loss function
- this forces the learning algorithm to not only fit the data but also keep the model weights as *small* as possible
- note that the regularization term should only be added to the loss function during training
- once the model is trained, we want to use the unregularized performance measure to evaluate the model's performance

- the hyperparameter λ controls how much we want to regularize the model
- if $\lambda = 0$, then ridge regression is just linear regression
- if λ is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean
- as with linear regression, we can perform ridge regression either by computing a closed-form equation or by performing gradient descent
- the pros and cons are the same; for gradient descent, we just add $\lambda \boldsymbol{w}$ to the gradient vector:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \frac{1}{n} \boldsymbol{X}^{\top} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}.$$

- it is important to *scale* the data before performing ridge regression, as it is sensitive to the scale of the input features, which is true of most regularized models

- on the other hand, we can assume a Laplace distribution for the weights, $\boldsymbol{w} \sim \text{Lap}(\boldsymbol{0}, b\boldsymbol{I})$
- the probability density for the Laplace distribution is given as:

$$p(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{1}{b}|x - \mu|\right).$$

- here, μ is a location parameter and $b > 0$ is a scale parameter
- the loss corresponding to the Laplace prior on the weights is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda||\boldsymbol{w}||_1,$$

  where $||\boldsymbol{w}||_1 = \sum_{j=0}^{d} |w_j|$ is the ℓ_1 norm of $\boldsymbol{w}$
- linear regression using this loss function is called *lasso regression*

- *Least Absolute Shrinkage and Selection Operator Regression* (usually simply called lasso regression) is another regularized version of linear regression: just like ridge regression, it adds a regularization term to the loss function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm
- an important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero)
- in other words, lasso regression automatically performs *feature selection* and outputs a *sparse model*
- that is, it often sets many weights to zero, effectively declaring the corresponding attributes to be completely irrelevant
- models that discard attributes can be easier for a human to understand, and may be less likely to overfit

- Figure 12 gives an intuitive explanation of why ℓ_1 regularization leads to weights of zero, while ℓ_2 regularization does not
- note that minimizing $\frac{1}{2n}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})+\frac{1}{2}\lambda||\boldsymbol{w}||_2^2$ or $\frac{1}{2n}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})+\lambda||\boldsymbol{w}||_1$ is equivalent to minimizing $\frac{1}{2n}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^\top(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})$ subject to the constraint that $\frac{1}{2}\lambda||\boldsymbol{w}||_2^2 \leq c$ or $\lambda||\boldsymbol{w}||_1 \leq c$, for some constant c that is related to λ
- now, in Figure 12(a) the diamond-shaped box represents the set of points $\boldsymbol{w}$ in two-dimensional weight space that have ℓ_1 norm less than c; our solution will have to be somewhere inside this box
- the concentric ovals represent contours of the mean squared error function, with the minimum loss at the center

- we want to find the point in the box that is *closest* to the minimum; we can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy
- and of course the corners are the points that have a value of zero in some dimension
- in Figure 12(b), we have done the same for the ℓ_2 norm, which represents a circle rather than a diamond
- here we can see that, in general, there is no reason for the intersection to appear on one of the axes; thus ℓ_2 regularization does not tend to produce zero weights

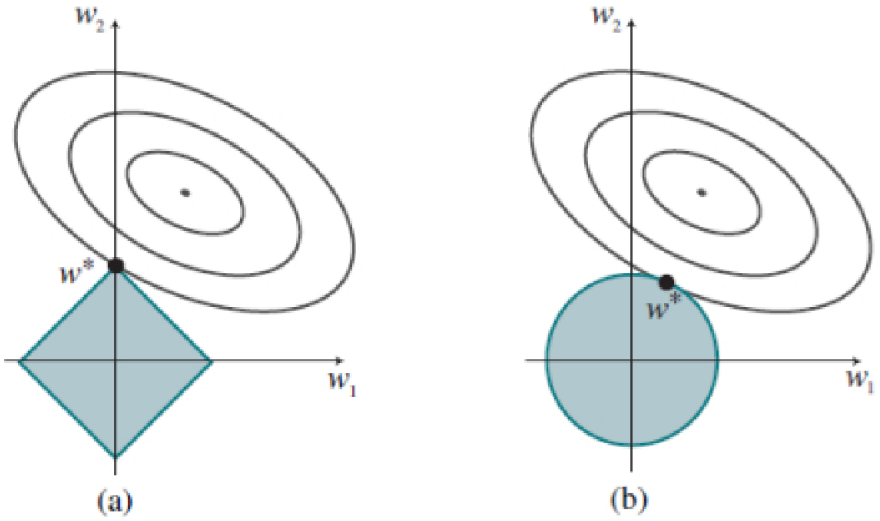


Figure 12: Why ℓ_1 regularization tends to produce a sparse model. Left: with ℓ_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: with ℓ_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

- the lasso loss function is not differentiable whenever $w_j = 0$ for some $j = 0, 1, \ldots, d$, but gradient descent still works fine if we use a *subgradient vector* $\boldsymbol{g}$ instead, when any $w_j = 0$
- we can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point
- a subgradient vector we can use for gradient descent with the lasso loss function is:

$$\boldsymbol{g}(\boldsymbol{w}) = \frac{1}{n}\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \text{sign}(\boldsymbol{w}),$$

where $$\text{sign}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \\ +1, & \text{if } x > 0 \end{cases}$$

- *elastic net* is a middle ground between ridge regression and lasso regression
- the regularization term is a simple mix of both ridge and lasso's regularization terms, and we can control the mix ratio $r \in [0, 1]$
- when $r = 0$, elastic net is equivalent to ridge regression, and when $r = 1$, it is equivalent to lasso regression:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\top}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + r\lambda||\boldsymbol{w}||_1 + \frac{1-r}{2}\lambda||\boldsymbol{w}||_2^2.$$

- so when should we use plain linear regression (i.e., without any regularization), ridge, lasso, or elastic net?
- it is almost always preferable to have at least a *little bit of regularization*, so generally we should avoid plain linear regression
- ridge is a good default, but if we suspect that only a few features are useful, we should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as we have discussed
- in general, elastic net is preferred over lasso, because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated

- a very different way to *regularize* iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum; this is called *early stopping*
- Figure 13 shows a complex model (in this case, a high-degree polynomial regression model) being trained with batch gradient descent
- as the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set
- after a while though, the validation error stops decreasing and starts to go back up
- this indicates that the model has started to overfit the training data

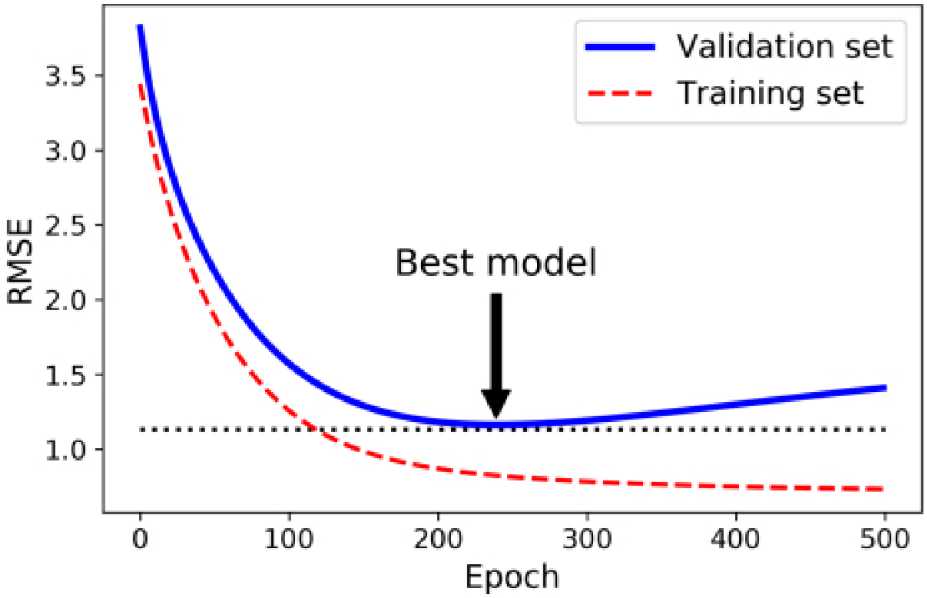


Figure 13: Early stopping regularization.

- with early stopping, we just stop training as soon as the validation error reaches the minimum
- with stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether we have reached the minimum or not
- one solution is to stop only after the validation error has been above the minimum for some time (when we are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum

1.6 Logistic regression

- some regression algorithms can be used for classification (and vice versa)
- *logistic regression* (also called *logit regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?)
- if the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”)
- this makes it a *binary classifier*

- just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly, like the linear regression model does, it outputs the *logistic* of this result:

$$\hat{p} = \sigma(\boldsymbol{w}^T \boldsymbol{x}).$$

- the logistic, denoted $\sigma(\cdot)$, is the *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1; it is defined as:

$$\sigma(t) = \frac{1}{1 + \exp(-t)},$$

and its graph is given in Figure 14

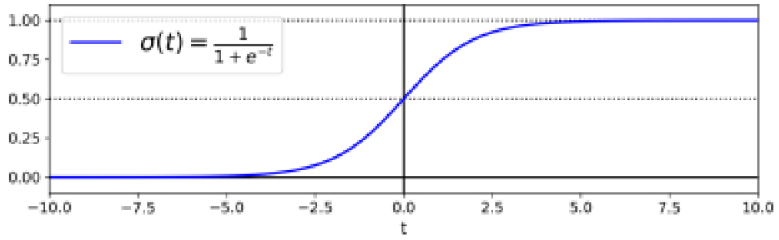


Figure 14: Sigmoid function.

- once the logistic regression model has estimated the probability $\hat{p}$ that an instance $\boldsymbol{x}$ belongs to the positive class, it can make its prediction $\hat{y}$ easily as:

$$\hat{y} = \begin{cases} 0, & \text{if } \hat{p} < 0.5 \\ 1, & \text{if } \hat{p} \geq 0.5 \end{cases}.$$

- notice that $\sigma(t) < 0.5$ when $t < 0$, and $\sigma(t) \geq 0.5$ when $t \geq 0$, so a logistic regression model predicts 1 if $\boldsymbol{w}^T \boldsymbol{x}$ is positive and 0 if it is negative
- the score t is often called the *logit*; the name comes from the fact that the *logit function*, defined as $\text{logit}(\hat{p}) = \log(\hat{p}/(1-\hat{p}))$, is the inverse of the sigmoid function
- indeed, if we compute the logit of the estimated probability $\hat{p}$, we will find that the result is t
- the logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class

- for training the logisitic regression, the objective is to set the weight vector $\boldsymbol{w}$ so that the model estimates high probabilities for positive instances ($y = 1$) and low probabilities for negative instances ($y = 0$)
- this idea is captured by the loss function shown below, for a single training instance $\boldsymbol{x}$:

$$l(\boldsymbol{w}) = \begin{cases} -\log \hat{p}, & \text{if } y = 1 \\ -\log(1 - \hat{p}), & \text{if } y = 0 \end{cases}.$$

- this loss function makes sense because $-\log t$ grows very large when t approaches 0, so the loss will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance
- on the other hand, $-\log t$ is close to 0 when t is close to 1, so the loss will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want

- it can be also written as:

$$l(\boldsymbol{w}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p}).$$

- the loss function over the whole training set is the average loss over all training instances
- it can be written in a single expression called the *log loss*, as:

$$\mathcal{L}(\boldsymbol{w}) = -\frac{1}{n} \sum_{i=1}^{n} \left[y^{(i)} \log \hat{p}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]. \tag{12}$$

- the bad news is that there is no known closed-form equation to compute the value of $\boldsymbol{w}$ that minimizes this loss function (there is no equivalent of the normal equation)
- the good news is that this loss function is convex, so gradient descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and we wait long enough)

- the gradient of the loss function is given by:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \frac{1}{n} \boldsymbol{X}^{\top} (\sigma(\boldsymbol{X}\boldsymbol{w}) - \boldsymbol{y}).$$

- this equation looks very much like equation (8): for each instance it computes the prediction error and multiplies it by the feature values, and then it computes the average over all training instances
- once we have the gradient vector containing all the partial derivatives, we can use it in the batch gradient descent algorithm
- for stochastic GD, we would take one instance at a time, and for mini-batch GD we would use a mini-batch at a time
- just like the other linear models, logistic regression models can be regularized using ℓ_1 or ℓ_2 penalties

- the logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers
- this is called *softmax regression*, or *multinomial logistic regression*
- assume we have a classification problem with q classes
- the idea is simple: when given an instance $\boldsymbol{x}$, the softmax regression model first computes a score $o_k(\boldsymbol{x})$ for each class $k = 1, 2, \ldots, q$, then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores
- the equation to compute $o_k(\boldsymbol{x})$ should look familiar, as it is just like the equation for linear regression prediction:

$$o_k(\boldsymbol{x}) = \boldsymbol{w}_k^\top \boldsymbol{x}.$$

- note that each class has its own dedicated parameter vector $\boldsymbol{w}_k \in \mathbb{R}^d$
- all these vectors are typically stored as columns in a parameter matrix $\boldsymbol{W} \in \mathbb{R}^{d \times q}$:

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \cdots & \boldsymbol{w}_q \end{bmatrix}.$$

- once we have computed the score of every class for the instance $\boldsymbol{x}$, we can estimate the probability $\hat{p}_k$ that the instance belongs to class k by running the scores through the softmax function:

$$\hat{p}_k = \text{softmax}(\boldsymbol{o}(\boldsymbol{x}))_k = \frac{\exp(o_k(\boldsymbol{x}))}{\sum_{l=1}^{q} \exp(o_l(\boldsymbol{x}))},$$

where

  - $\boldsymbol{o}(\boldsymbol{x})$ is a vector containing the scores of each class for the instance $\boldsymbol{x}$
  - $\text{softmax}(\boldsymbol{o}(\boldsymbol{x}))_k$ is the estimated probability that the instance $\boldsymbol{x}$ belongs to class k, given the scores of each class for that instance

- the function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials)
- the scores are generally called *logits* or *log-odds* (although they are actually *unnormalized log-odds*)
- just like the logistic regression classifier, the softmax regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score):

$$\hat{y} = \underset{k}{\operatorname{argmax}} \operatorname{softmax}(\boldsymbol{o}(\boldsymbol{x}))_k = \underset{k}{\operatorname{argmax}} o_k(\boldsymbol{x}) = \underset{k}{\operatorname{argmax}} \boldsymbol{w}_k^\top \boldsymbol{x}.$$

- the argmax operator returns the value of a variable that maximizes a function
- in this equation, it returns the value of k that maximizes the estimated probability $\operatorname{softmax}(\boldsymbol{o}(\boldsymbol{x}))_k$
- the softmax regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants, not multiple people in one picture

- for training, the objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes)
- minimizing the loss function shown below, called the *cross entropy*, should lead to this objective, because it penalizes the model when it estimates a low probability for a target class
- cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes

$$\mathcal{L}(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{q} y_k^{(i)} \log \hat{p}_k^{(i)},$$

where

- $y_k^{(i)}$ is the target probability that the ith instance belongs to class k. In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

- notice that when there are just two classes ($q = 2$), this loss function is equivalent to the logistic regression's loss function in (12)

- the cross entropy between probability distributions p and q, denoted $H(p, q)$, measures the *mismatch* between probabilities q upon seeing data that were actually generated according to probabilities p:

$$H(p, q) = -\sum_x p(x) \log q(x).$$

- the lowest possible cross entropy is achieved when $p = q$; in this case, the cross-entropy between p and q is:

$$H(p, p) = H(p) = -\sum_x p(x) \log p(x),$$

  i.e., cross entropy is equal to the entropy of probability distribution p
- the gradient of the cross entropy loss function with regard to $\boldsymbol{w}_k$ is given by:

$$\nabla_{\boldsymbol{w}_k} \mathcal{L}(\boldsymbol{W}) = \frac{1}{n} \boldsymbol{X}^\top (\hat{\boldsymbol{p}}_k - \boldsymbol{y}_k) = \frac{1}{n} \boldsymbol{X}^\top (\text{softmax}(\boldsymbol{o}(\boldsymbol{x}))_k - \boldsymbol{y}_k).$$

- now we can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix $\boldsymbol{W}$ that minimizes the loss function

Thank you!