Computer Vision

Lecture 04
Optimization Problem and Gradient Descent
Military College of Signals

Asim D. Bakhshi
asim.dilawar@mcs.edu.pk

May 11, 2022

Contents

1 Introduction

In the last lecture we introduced two key components in the context of the image classification task:

- A (parameterized) **score function** mapping the raw image pixels to class scores (e.g. a linear function).
- A **loss function** that measured the quality of a particular set of parameters based on how well the induced scores agreed with the ground truth labels in the training data. We saw that there are many ways and versions of this (e.g. Softmax/SVM).

Concretely, recall that the linear function had the form $f(x_i, W) = Wx_i$ and the SVM we developed was formulated as:

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [\max(0, f(x_i; \mathbf{W})_j - f(x_i; W)_{y_i} + 1)] + \alpha R(\mathbf{W})$$

We saw that a setting of the parameters $\mathbf{W}$ that produced predictions for examples x_i consistent with their ground truth labels y_i would also have a very low loss L. We are now going to introduce the third and last key component: **optimization**. Optimization is the process of finding the set of parameters $\mathbf{W}$ that minimize the loss function.

Foreshadowing. Once we understand how these three core components interact, we will revisit the first component (the parameterized function mapping) and extend it to functions much more complicated than a linear mapping: First entire Neural Networks, and then Convolutional Neural Networks. The loss functions and the optimization process will remain relatively unchanged.

2 Visualizing the Loss Function

The loss functions we come across in visual recognition problems are usually defined over very high-dimensional spaces (e.g. in CIFAR-10 a linear classifier weight matrix is of size $[10 \times 3073]$ for a total of 30730 parameters), making them difficult to visualize. However, we can still gain some intuitions about one by slicing through the high-dimensional space along rays (1 dimension), or along planes (2 dimensions). For example, we can generate a random weight matrix $\mathbf{W}$ (which corresponds to a single point in the space), then march along a ray and record the loss function value along the way. That is, we can generate a random direction $\mathbf{W}_1$ and compute the loss along this direction by evaluating $L(\mathbf{W} + a\mathbf{W}_1)$ for different values of a. This process generates a simple plot with the value of a as the x-axis and the value of the loss function as the y-axis. We can also carry out the same procedure with two dimensions by evaluating the loss $L(\mathbf{W} + a\mathbf{W}_1 + b\mathbf{W}_2)$ as we vary a, b. In a plot, a, b could then correspond to the x-axis and the y-axis, and the value of the loss function can be visualized with a color as shown in Figure 1.

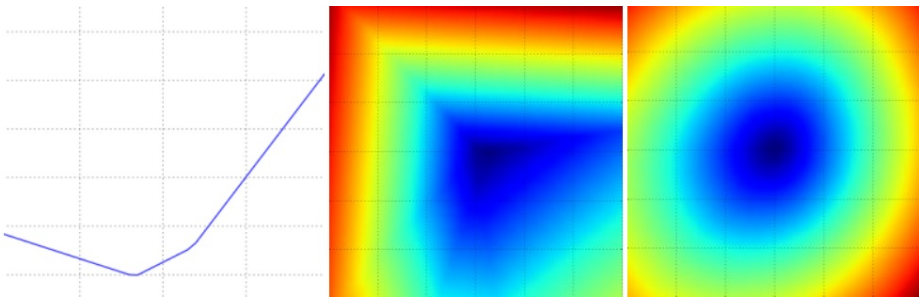

Figure 1: Loss function landscape for the Multiclass SVM (without regularization) for one single example (left,middle) and for a hundred examples (right) in CIFAR-10. Left: one-dimensional loss by only varying a. Middle, Right: two-dimensional loss slice, Blue = low loss, Red = high loss. Notice the piecewise-linear structure of the loss function. The losses for multiple examples are combined with average, so the bowl shape on the right is the average of many piece-wise linear bowls (such as the one in the middle).

2.1 Interpretation of Loss Function Landscape

We can explain the piecewise-linear structure of the loss function by examining the math. For a single example we have:

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

It is clear from the equation that the data loss for each example is a sum of (zero-thresholded due to the $\max(0, -)$ function) linear functions of $\mathbf{W}$. Moreover, each row of $\mathbf{W}$ (i.e. $\mathbf{w}_j$) sometimes has a positive sign in front of it (when it corresponds to a wrong class for an example), and sometimes a negative sign (when it corresponds to the correct class for that example). To make this more explicit, consider a simple dataset that contains three 1-dimensional points and three classes. The full SVM loss (without regularization) becomes:

$$\begin{aligned}
L_0 &= \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1) \\
L_1 &= \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1) \\
L_2 &= \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1) \\
L &= (L_0 + L_1 + L_2)/3
\end{aligned}$$

Since these examples are 1-dimensional, the data x_i and weights w_j are numbers. Looking at, for instance, w_0, some terms above are linear functions of w_0 and each is clamped at zero. We can visualize this as in Figure 2

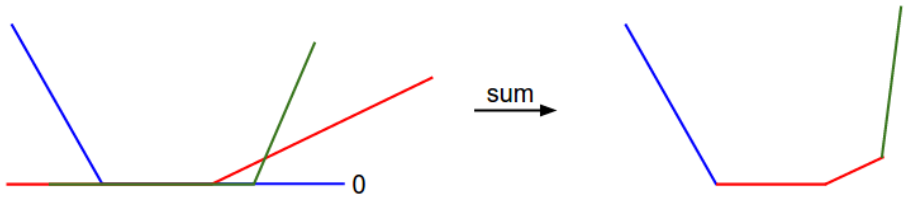

Figure 2: 1-dimensional illustration of the data loss. The x-axis is a single weight and the y-axis is the loss. The data loss is a sum of multiple terms, each of which is either independent of a particular weight, or a linear function of it that is thresholded at zero. The full SVM data loss is a 30,730-dimensional version of this shape.

As an aside, you may have guessed from its bowl-shaped appearance that the SVM cost function is an example of a convex function. There is a large amount of literature devoted to efficiently minimizing these types of functions,

called convex optimization [1]. Once we extend our score functions f to Neural Networks our objective functions will become non-convex, and the visualizations above will not feature bowls but complex, bumpy terrains.

Non-differentiable loss functions. As a technical note, you can also see that the kinks in the loss function (due to the max operation) technically make the loss function non-differentiable because at these kinks the gradient is not defined. However, the subgradient still exists and is commonly used instead. In this class will use the terms subgradient and gradient interchangeably.

3 Optimization

To reiterate, the loss function lets us quantify the quality of any particular set of weights **W**. The goal of optimization is to find **W** that minimizes the loss function. We will now motivate and slowly develop an approach to optimizing the loss function.

3.1 Strategy 1: Random Search (Bad Idea)

Since it is so simple to check how good a given set of parameters **W** is, the first (very bad) idea that may come to mind is to simply try out many different random weights and keep track of what works best.

If we would practically implement it, we'll have to try out several random weight vectors **W**, and some of them work better than others. We can take the best weights **W** found by this search and try it out on the test set.

With the best **W** this gives an accuracy of about 12-16%. Given that guessing classes completely at random achieves only 10%, that's not a very bad outcome for a such a brain-dead random search solution!

Core idea: iterative refinement. Of course, it turns out that we can do much better. The core idea is that finding the best set of weights **W** is a very difficult or even impossible problem (especially once **W** contains weights for entire complex neural networks), but the problem of refining a specific set of weights W to be slightly better is significantly less difficult. In other words, our approach will be to start with a random **W** and then iteratively refine it, making it slightly better each time.

Our strategy will be to start with random weights and iteratively refine them over time to get lower loss

Blindfolded hiker analogy. One analogy that you may find helpful going forward is to think of yourself as hiking on a hilly terrain with a blindfold on, and trying to reach the bottom. In the example of CIFAR-10, the hills are 30730-dimensional, since the dimensions of **W** are 10×3073. At every point on the hill we achieve a particular loss (the height of the terrain).

3.2 Strategy 2: Random Local Search

The first strategy you may think of is to try to extend one foot in a random direction and then take a step only if it leads downhill. Concretely, we will start out with a random $\mathbf{W}$, generate random perturbations $\delta\mathbf{W}$ to it and if the loss at the perturbed $\mathbf{W} + \delta\mathbf{W}$ is lower, we will perform an update.

Using the same number of loss function evaluations as before (1000), this approach achieves test set classification accuracy of 20-25% with lots of random initialization. This is better, but still wasteful and computationally expensive.

3.3 Strategy 3: Follow the Gradient

In the previous section we tried to find a direction in the weight-space that would improve our weight vector (and give us a lower loss). It turns out that there is no need to randomly search for a good direction: we can compute the best direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descent (at least in the limit as the step size goes towards zero). This direction will be related to the gradient of the loss function. In our hiking analogy, this approach roughly corresponds to feeling the slope of the hill below our feet and stepping down the direction that feels steepest.

In one-dimensional functions, the slope is the instantaneous rate of change of the function at any point you might be interested in. The gradient is a generalization of slope for functions that don't take a single number but a vector of numbers. Additionally, the gradient is just a vector of slopes (more commonly referred to as derivatives) for each dimension in the input space. The mathematical expression for the derivative of a 1-D function with respect its input is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1}$$

When the functions of interest take a vector of numbers instead of a single number, we call the derivatives partial derivatives, and the gradient is simply the vector of partial derivatives in each dimension.

4 Mathematical Review of Gradients

4.1 Partial Derivatives

For functions with multiple inputs, we must make use of the concept of partial derivatives. The partial derivative $\frac{\partial f(\mathbf{x})}{\partial x_i}$ measures how f changes as only the variable x_i increases at point x.

4.2 Gradients

A gradient is a multidimensional generalization of a derivative. Suppose you have a function $f : \mathbb{R}^D \to \mathbb{R}$ that takes a vector $\langle x = x_1, x_2, \cdots, x_D \rangle$ as input and produces a scalar value as output. You can differentiate this function according to any one of the inputs; for instance, you can compute $\frac{\partial f}{\partial x_5}$ to get the derivative with respect to the fifth input. The gradient of f is just the vector consisting of the derivative f with respect to each of its input coordinates independently, and is denoted ∇f, or, when the input to f is ambiguous, $\nabla_x f$. This is defined as:

$$\nabla_x f = \left\langle \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_D} \right\rangle \tag{2}$$

As an example, consider the function $f(x_1, x_2, x_3) = x_1^3 + 5x_1x_2 - 3x_2x_3^2$. The gradient is:

$$\nabla_x f = \langle 3x_1^2 + 5x_2, 5x_1 - 3x_3^2, -6x_2x_3 \rangle$$

Note that if $f : \mathbb{R}^D \to \mathbb{R}$, then $\nabla f : \mathbb{R}^D \to \mathbb{R}^D$. If you evaluate $\nabla f(x)$, this will give you the gradient at x, a vector in $\mathbb{R}^D$. This vector can be interpreted as the direction of ***steepest ascent***: namely, if you were to travel an infinitesimal amount in the direction of the gradient, you would go uphill (i.e., increase f) the most.

4.3 Directional Derivatives

The **directional derivative** in direction u (a unit vector) is the slope of the function f in direction u. In other words, the directional derivative is the derivative of the function $f(\boldsymbol{x} + \alpha \boldsymbol{u})$ with respect to α, evaluated at $\alpha = 0$. Using the chain rule, we can see that $\frac{\partial f(\boldsymbol{x} + \alpha \boldsymbol{u})}{\partial \alpha}$ evaluates to $\boldsymbol{u}^{\mathrm{T}} \nabla_x f(\boldsymbol{x})$ when $\alpha = 0$.

5 Gradient Descent

5.1 Casting the Optimization Problem

Suppose you are trying to find the maximum of a function $f(x)$. The optimizer maintains a current estimate of the parameter of interest, x. At each step, it measures the **gradient** of the function it is trying to optimize. This measurement occurs at the current location, x. Call the gradient ∇. It then takes a step in the direction of the gradient, where the size of the step is controlled by a parameter η. The complete step is denoted by $x \leftarrow x + \eta \nabla$. This is the basic idea of **gradient ascent** shown in Figure 3.

The opposite of gradient ascent is **gradient descent**. Here, all of our learning problems will be framed as minimization problems (trying to reach the bottom of a ditch, rather than the top of a hill). Therefore, descent is the primary

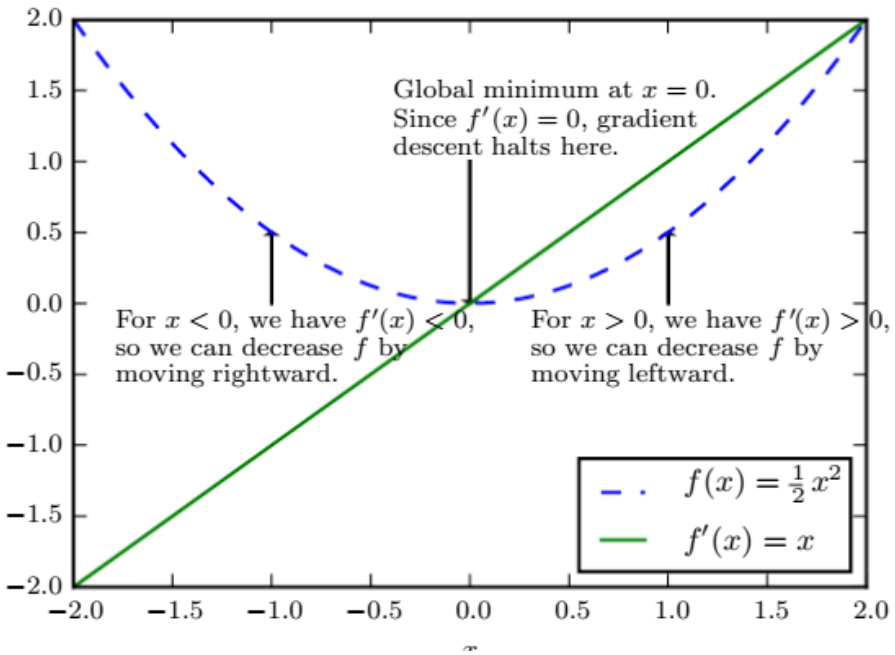

Figure 3: An illustration of how the gradient descent algorithm uses the derivatives of a function can be used to follow the function downhill to a minimum.

approach you will use. One of the major conditions for gradient ascent being able to find the true, global minimum, of its objective function is convexity. Without convexity, all is lost.

Recall that the function we want to minimize or maximize is called the **objective function** or **criterion**. When we are minimizing it, we may also call it the **cost function**, **loss function**, or **error function**. When gradient of our loss function $f'(x) = 0$, the derivative provides no information about which direction to move.

Critical Points. The points where $f'(x) = 0$ are known as **critical points** or **stationary points**. A **local minimum** is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A **local maximum** is a point where $f(x)$ is higher than at all neighboring points. So it is not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither **maxima** nor **minima**. These are known as **saddle points**. See Figure 4 for examples of each type of critical point.

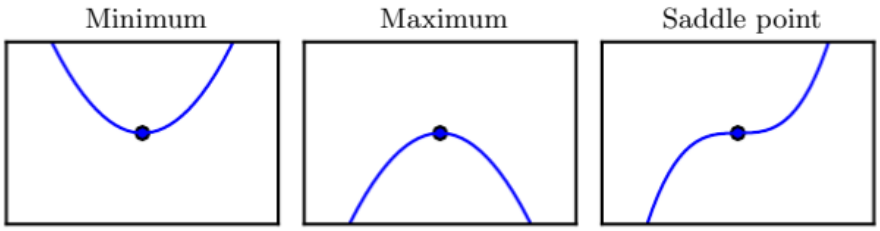

Figure 4: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors that are both higher and lower than the point itself.

A point that obtains the absolute lowest value of $f(x)$ is a **global minimum**. It is possible for there to be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of computer vision, we optimize functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions. All of this makes optimization very difficult, especially when the input to the function is multidimensional. We therefore usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense. See Figure 3 for an example.

To minimize f, we would like to find the direction in which f decreases the fastest. We can decrease f by moving in the direction of the negative gradient. This is known as the method of **steepest descent** or **gradient descent**. Steepest descent proposes a new point

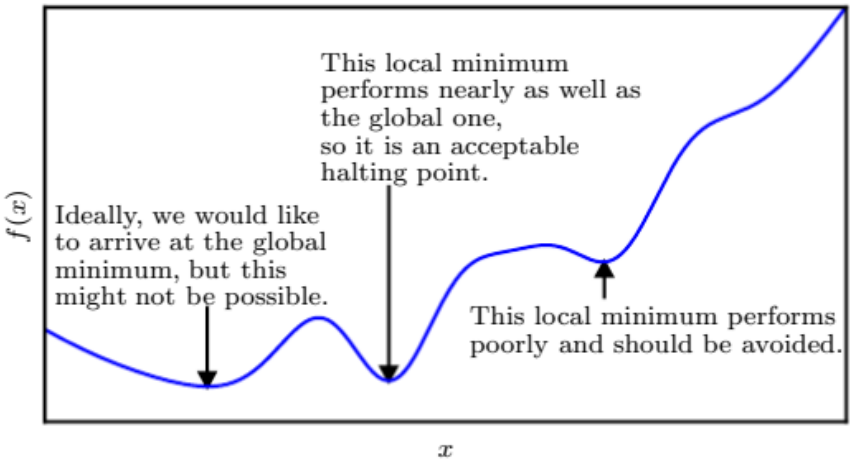

Figure 5: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

$$x' = x - \alpha \nabla_x f(x) \tag{3}$$

where α is the **learning rate**, a positive scalar determining the size of the step. We can choose in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(x - \nabla_x f(x))$ for several values of and choose the one that results in the smallest objective function value. This last strategy is called a **line search**.

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm, and just jump directly to the critical point by solving the equation

$$\nabla_x f(x) = 0 \tag{4}$$

for every x.

Although gradient descent is limited to optimization in continuous spaces, the general concept of repeatedly making a small move (that is approximately the best small move) towards better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called **hill climbing**.

5.2 The Algorithm

The gradient descent algorithm initializes the variable to be optimized (in our case the weight vector $\mathbf{w}^{(0)}$), takes following inputs:

1. The function $f(\mathbf{x}, \mathbf{w})$ to be minimized.
2. The number of iterations K to run.
3. A sequence of learning rates $\alpha_1, \alpha_2, \cdots, \alpha_K$ to address the case that you might want to start your mountain climbing taking large steps, but only take small steps when you are close to the peak.

The algorithm then performs following two steps in each iteration k:

1. Compute the gradient $\nabla^{(k)} \leftarrow \nabla_x f(\mathbf{x}, \mathbf{w})$.
2. Take a step $\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k-1)} - \alpha^{(k)} \nabla^{(k)}$ down the gradient.

After the iterative process completes, the algorithm returns the updated weight vector $\mathbf{w}^{(K)}$ after K iterations.

Note that the only real work you need to do to apply a gradient descent method is be able to compute derivatives.

6 Implementation of Gradient Descent

Recall our **regularized objective function** of linear classification problem from last lecture

$$L' = L + \lambda R(\mathbf{w}) \tag{5}$$

where L was the squared loss function and $\lambda R(\mathbf{w})$ was the regularizing terms. Note that the general form of (4) is

$$\text{(Objective Function)} = \text{(Loss Function)} + \text{(Regularizer)} \tag{6}$$

where choices of loss function and regularizer would dictate complexity, tractability and choice of solutions to our optimization problem. Also remember that our linear model was of the form

$$y = f(x; w_0; w_1) = w_0 + w_1 x \tag{7}$$

and average squared loss was of the form

$$L = \frac{1}{N} \sum_{i=1}^{N} \left(\left(w_0 + w_1 x^{(i)} \right) - y^{(i)} \right)^2 \tag{8}$$

For linear classification with gradient descent optimization, we can now recast (8) as

$$L(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\left(h_{\mathbf{w}}\left(x^{(i)}\right) - y^{(i)}\right)^2 \tag{9}$$

where $h_{\mathbf{w}}\left(x^{(i)}\right) = w_0 + w_1 x^{(i)}$ a linear hypothesis function. Our aim is to find values of w_0, w_1 which provide the best fit of our hypothesis to a training set. The training set examples are labeled x, y, where x is the input value and y is the output. The ith training example is labeled as $x^{(i)}$,$y^{(i)}$.

The loss (or cost) function $L(\mathbf{w})$ measures the average amount that the model's predictions vary from the correct values, hence we can think of it as a measure of the model's performance on the training set. The cost is higher when the model is performing poorly on the training set. The objective of the learning algorithm, then, is to find the parameters $\mathbf{w}$ which give the minimum possible loss $L(\mathbf{w})$.

6.1 Example 1: Single Variable Case

We want to utilize the gradient descent approach to find $\mathbf{w}$ that minimizes the loss $L(\mathbf{w})$. But let's forget the little complex loss function of (9) for a moment and look at gradient descent as a minimization technique in general. Let us take a much simpler function

$$L(w) = w^2 \tag{10}$$

and try to use gradient descent to find w that minimize $L(w)$. We can now set our inputs to the gradient descent algorithm (section 5.2) as follows:

1. $w^{(0)} = 3$
2. $K = 10$
3. $\alpha = 0.1$

6.2 Example 2: Multivariate Case

Our loss function (9) is multivariate so let's do a simple multivariate problem given by the cost function

$$L(\mathbf{w}) = w_1^2 + w_2^2 \tag{11}$$

When there are multiple variables in the minimization objective, gradient descent defines a separate update rule for each variable. The update rule for w_1 would use the partial derivative of $L(\mathbf{w})$ with respect to w_1 and the update rule for w_2 would use the partial derivative of $L(\mathbf{w})$ with respect to w_2. The objective of our minimization problem is now constructed as

$$\min_{w_1, w_2} L(w_1, w_2)$$

You now have to write following:

1. Update rules for both parameters w_1 and w_2.

2. Partial derivatives.

Note that when implementing the update rule in software, w_1 and w_2 should not be updated until after you have computed the new values for both of them. Specifically, you don't want to use the new value of w_1 to calculate the new value of w_2.

6.3 Gradient Descent for Euclidean Squared Loss Function

Now that we know how to perform gradient descent on an equation with multiple variables, we can return to looking at gradient descent on our loss function of (9). Taking the derivative of this equation is a little more tricky. The key thing to remember is that x and y are not variables for the sake of the derivative. Rather, they represent a large set of constants (your training set). So while taking the derivative of the cost function, we'll treat x and y like we would any other constant.

$$\frac{\partial}{\partial \mathbf{w}} L\left(\mathbf{w}\right) = \frac{\partial}{\partial \mathbf{w}} \left(\frac{1}{N} \sum_{i=1}^{N} \left(h_{\mathbf{w}} \left(x^{(i)} \right) - y^{(i)} \right)^2 \right) \tag{12}$$

7 Computing the Gradient

There are two ways to compute the gradient: A slow, approximate but easy way (numerical gradient), and a fast, exact but more error-prone way that requires calculus (analytic gradient).

7.1 Numerical Computation of the Gradient

We can use (1) to compute the gradient numerically. However, we must note that in the mathematical formulation the gradient is defined in the limit as h goes towards zero, but in practice it is often sufficient to use a very small value (such as $1e-5$). Ideally, you want to use the smallest step size that does not lead to numerical issues. Additionally, in practice it often works better to compute the numeric gradient using the centered difference formula given by:

$$\frac{f(x+h) - f(x-h)}{2h} \tag{13}$$

Effect of Step Size (Learning Rate). The gradient tells us the direction in which the function has the steepest rate of increase, but it does not tell us how far along this direction we should step. As we will see later in the course, choosing the step size (also called the learning rate) will become one of the most important (and most headache-inducing) hyperparameter settings in

training a neural network. In our blindfolded hill-descent analogy, we feel the hill below our feet sloping in some direction, but the step length we should take is uncertain. If we shuffle our feet carefully we can expect to make consistent but very small progress (this corresponds to having a small step size). Conversely, we can choose to make a large, confident step in an attempt to descend faster, but this may not pay off (see Figure 6) .

A Problem of Efficiency. You may have noticed that evaluating the numerical gradient has complexity linear in the number of parameters. In our example we had 30730 parameters in total and therefore had to perform 30731 evaluations of the loss function to evaluate the gradient and to perform only a single parameter update. This problem only gets worse, since modern Neural Networks can easily have tens of millions of parameters. Clearly, this strategy is not scalable and we need something better.

7.2 Analytical Computation of the Gradient

The numerical gradient is very simple to compute using the finite difference approximation, but the downside is that it is approximate (since we have to pick a small value of h, while the true gradient is defined as the limit as h goes to zero), and that it is very computationally expensive to compute. The second way to compute the gradient is analytically using Calculus, which allows us to derive a direct formula for the gradient (no approximations) that is also very fast to compute. However, unlike the numerical gradient it can be more error prone to implement, which is why in practice it is very common to compute the analytic gradient and compare it to the numerical gradient to check the correctness of your implementation. This is called a gradient check.

Lets use the example of the SVM loss function for a single datapoint:

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \right] \tag{14}$$

We can differentiate the function in (3) with respect to the weights. For example, taking the gradient with respect to w_{y_i} we obtain:

$$\nabla_{w_{y_i}} L_i = - \left(\sum_{j \neq y_i} \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i \tag{15}$$

where $\mathbf{1}$ is the indicator function that is one if the condition inside is true or zero otherwise. While the expression may look scary when it is written out, when you're implementing this in code you'd simply count the number of classes that didn't meet the desired margin (and hence contributed to the loss function) and then the data vector x_i scaled by this number is the gradient. Notice that this is the gradient only with respect to the row of $\mathbf{W}$ that corresponds to the correct class. For the other rows where $j \neq y_i$ the gradient is:

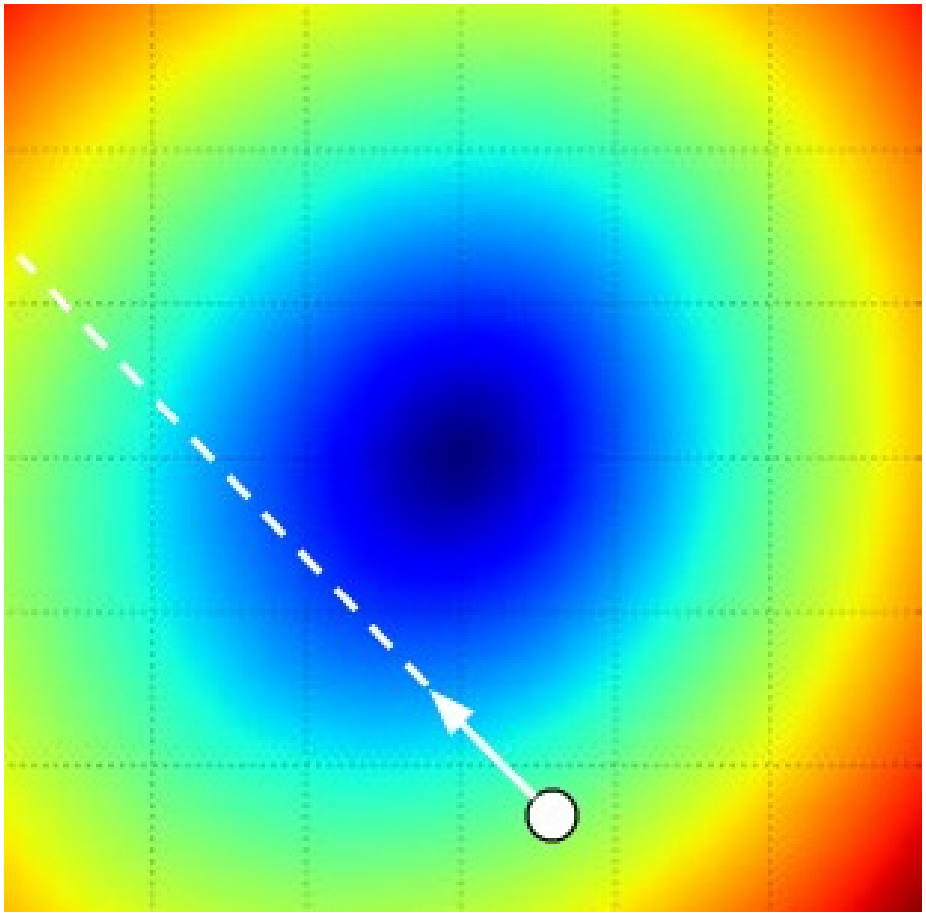

Figure 6: Visualizing the effect of step size. We start at some particular spot W and evaluate the gradient (or rather its negative - the white arrow) which tells us the direction of the steepest decrease in the loss function. Small steps are likely to lead to consistent but slow progress. Large steps can lead to better progress but are more risky. Note that eventually, for a large step size we will overshoot and make the loss worse. The step size (or as we will later call it - the learning rate) will become one of the most important hyperparameters that we will have to carefully tune.

$$\nabla_{w_j} L_i = \mathbf{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i \tag{16}$$

Once you derive the expression for the gradient it is straight-forward to implement the expressions and use them to perform the gradient update.

8 Variants of Gradient Descent Optimization

8.1 Mini-Batch Gradient Descent

In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data [2]. For example, in current state of the art ConvNets, a typical batch contains 256 examples from the entire training set of 1.2 million. This batch is then used to perform a parameter update.

The reason this works well is that the examples in the training data are correlated. To see this, consider the extreme case where all 1.2 million images in ILSVRC are in fact made up of exact duplicates of only 1000 unique images (one for each class, or in other words 1200 identical copies of each image). Then it is clear that the gradients we would compute for all 1200 identical copies would all be the same, and when we average the data loss over all 1.2 million images we would get the exact same loss as if we only evaluated on a small subset of 1000. In practice of course, the data-set would not contain duplicate images, the gradient from a mini-batch is a good approximation of the gradient of the full objective. Therefore, much faster convergence can be achieved in practice by evaluating the mini-batch gradients to perform more frequent parameter updates.

8.2 Stochastic Gradient Descent

The extreme case of this is a setting where the mini-batch contains only a single example. This process is called Stochastic Gradient Descent (SGD) [3] (or also sometimes on-line gradient descent). This is relatively less common to see because in practice due to vectorized code optimizations it can be computationally much more efficient to evaluate the gradient for 100 examples, than the gradient for one example 100 times. Even though SGD technically refers to using a single example at a time to evaluate the gradient, you will hear people use the term SGD even when referring to mini-batch gradient descent (i.e. mentions of MGD for "Mini-batch Gradient Descent", or BGD for "Batch gradient descent" are rare to see), where it is usually assumed that mini-batches are used. The size of the mini-batch is a hyper-parameter but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

9 Summary

1. We developed the intuition of the loss function as a high-dimensional optimization landscape in which we are trying to reach the bottom. The working analogy we developed was that of a blindfolded hiker who wishes to reach the bottom. In particular, we saw that the SVM cost function is piece-wise linear and bowl-shaped.

2. We motivated the idea of optimizing the loss function with iterative refinement, where we start with a random set of weights and refine them step by step until the loss is minimized.

3. We saw that the gradient of a function gives the steepest ascent direction and we discussed a simple but inefficient way of computing it numerically using the finite difference approximation (the finite difference being the value of h used in computing the numerical gradient).

4. We saw that the parameter update requires a tricky setting of the step size (or the learning rate) that must be set just right: if it is too low the progress is steady but slow. If it is too high the progress can be faster, but more risky. We will explore this trade-off in much more detail in future sections.

5. We discussed the trade-offs between computing the numerical and analytic gradient. The numerical gradient is simple but it is approximate and expensive to compute. The analytic gradient is exact, fast to compute but more error-prone since it requires the derivation of the gradient with math. Hence, in practice we always use the analytic gradient and then perform a gradient check, in which its implementation is compared to the numerical gradient.

6. We introduced the Gradient Descent algorithm which iteratively computes the gradient and performs a parameter update in loop.

What is coming up next: The core takeaway from this lecture is that the ability to compute the gradient of a loss function with respect to its weights (and have some intuitive understanding of it) is the most important skill needed to design, train and understand neural networks. In the next section we will develop proficiency in computing the gradient analytically using the chain rule, otherwise also referred to as backpropagation. This will allow us to efficiently optimize relatively arbitrary loss functions that express all kinds of Neural Networks, including Convolutional Neural Networks.

References

[1] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[2] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent". *Cited on* 14, p. 2, 2012.

[3] Léon Bottou. "Stochastic gradient descent tricks". In: *Neural networks: Tricks of the trade*. Springer, 2012. Pp. 421–436.