Image Classification

Optimization

Recap

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \overset{\text{e.g.}}{=} Wx$
- We have a **loss function**:

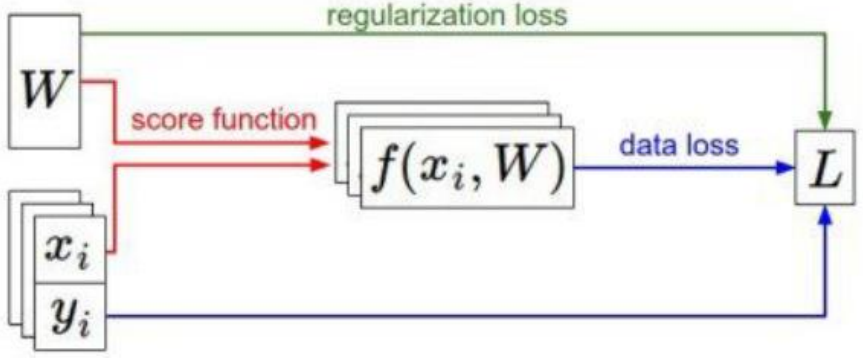
Softmax

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

SVM

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$ Full loss



Optimization

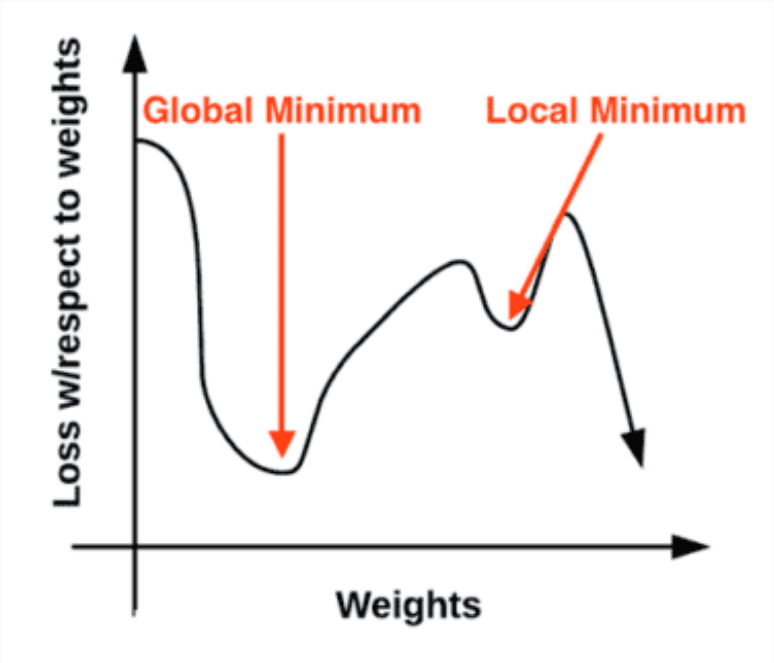
- We saw that a setting of the parameters W that produced predictions for examples x_i consistent with their ground truth labels y_i would also have a very low loss L

- We are now going to introduce the third and last key component: **optimization**.

- Optimization is the process of finding the set of parameters W that minimize the loss function.

Loss function visualization

- The loss functions we'll look at in this class are usually defined over very high-dimensional spaces (e.g. in CIFAR-10 a linear classifier weight matrix is of size [10 x 3073] for a total of 30,730 parameters), making them difficult to visualize.



Strategy #1: A first very bad idea solution: Random search

- 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.

- Instead of relying on pure randomness, we need to define an optimization algorithm that allows us to literally improve W and b.

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.

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 W , generate random perturbations δW to it and if the loss at the perturbed $W + \delta W$ is lower, we will perform an update.

Strategy #3: Following the Gradient

- 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
- 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.

Computing the gradient

- The mathematical expression for the **derivative** of a **1-D** function with respect its input is:

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

- In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

- The slope in any direction is the dot product of the direction with the gradient

- The direction of steepest descent is the negative gradient

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (first dim)**:**

[0.34 + **0.0001**,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

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

?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (second dim)**:**

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,

(1.25353 - 1.25347)/0.0001
= 0.6

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

?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,…]

(1.25347 - 1.25347)/0.0001
= 0

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

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

W + h (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,…]

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

Computing the gradient analytically with Calculus

- 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 30,731 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.

- The loss is just a function of W, Use calculus to compute an analytic gradient

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

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

dW = ...
(some function
data and W)

gradient dW:

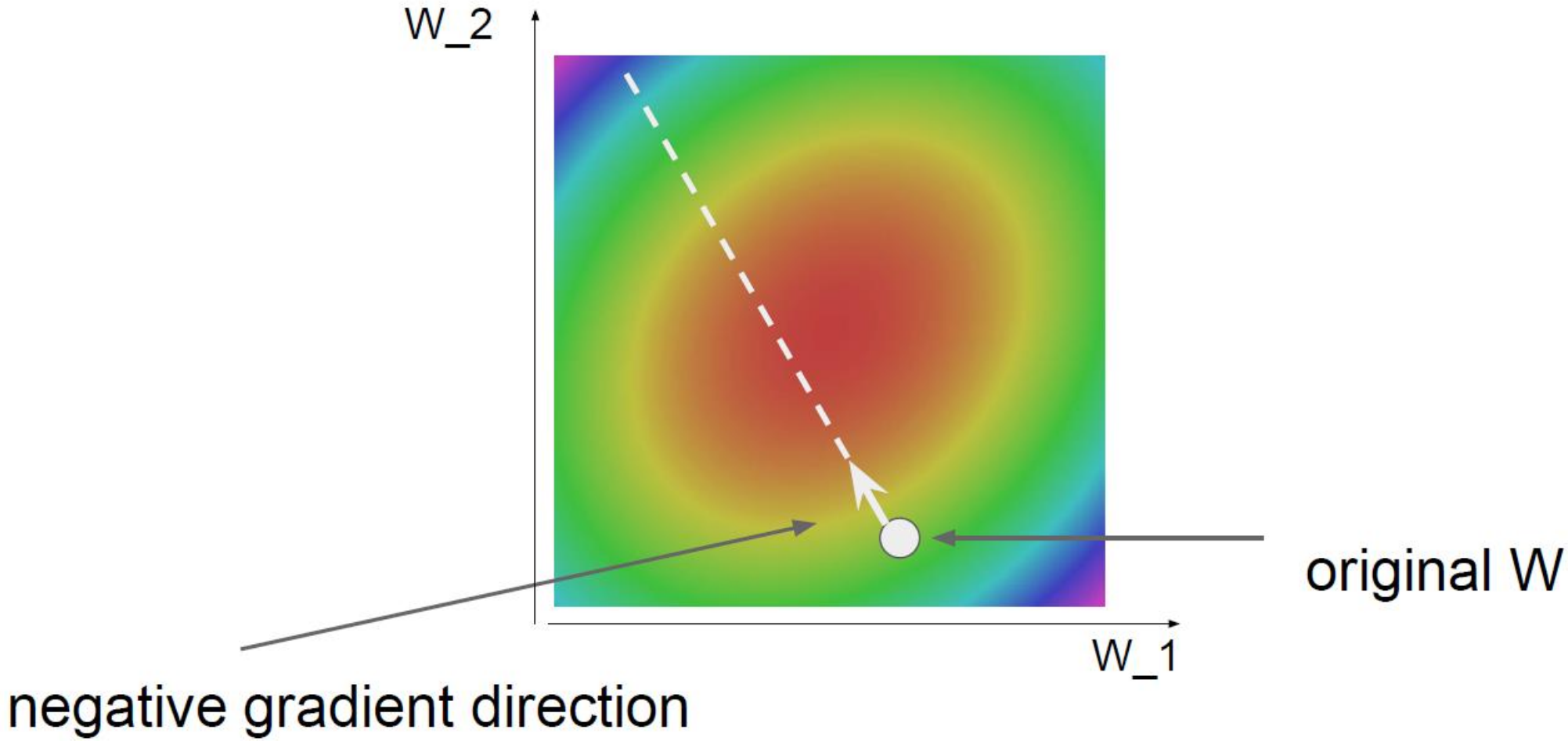
[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

Gradient Decent

- Initialize the weights
- Evaluate the gradient for each weight
- Upgrade the weight with a step size in the negative direction of the gradient

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```


W_2
W_1
original W
negative gradient direction

Step size

- 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.

Mini-batch gradient descent
Stochastic Gradient Descent (SGD)

- 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. This batch is then used to perform a parameter update.

- This is commonly referred to **as stochastic gradient decent (SGD).**

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Some common optimization techniques

- SGD with Momentum
- RMSprop
- Adam (Adaptive Moment Estimation)
- …

To sum up …

- The loss function represents an optimization landscape where we aim to reach the bottom.
- **Iterative refinement** is employed to optimize the loss function by gradually adjusting weights until the loss is minimized.
- The **gradient** of a function indicates the steepest ascent direction.
- Numerical gradients are simple but approximate and computationally expensive. Analytic gradients are exact but require mathematical derivation, making them more error-prone.
- Setting the step size (learning rate) for parameter updates is crucial.
- The **Gradient Descent** algorithm was introduced as an iterative process that computes gradients and updates parameters to minimize the loss.

- The core takeaway from this 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.

Web demo for a linear classifier

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

Linear classifier for image classification

- Taking the raw pixels and feeding them to the linear classifier in not such a great idea given the high dimensionality of the problem.

- What was common before the dominance of neural networks and deep learning was a two-stage approach.

- First, you would talk the image and compute different feature representations such as (Ex: SIFT, SURF, HOG, Bag of words, …)

- And then those features are given to the classifier not the entire image