Lecture 6: Scaling to complex models by learning with optimization algorithms

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

- Gradient descent
- Convex optimization as a simple model
- Conditioning

In [1]:
```
import numpy
import scipy
import matplotlib
import time

def hide_code_in_slideshow():
    from IPython import display
    import binascii
    import os
    uid = binascii.hexlify(os.urandom(8)).decode()
    html = """<div id="%s"></div>
    <script type="text/javascript">
        $(function(){
            var p = $("#%s");
            if (p.length==0) return;
            while (!p.hasClass("cell")) {
                p=p.parent();
                if (p.prop("tagName") =="body") return;
            }
            var cell = p;
            cell.find(".input").addClass("hide-in-slideshow")
        });
    </script>""" % (uid, uid)
    display.display_html(html, raw=True)
```

Review: The Empirical Risk

Suppose we have a dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where $x_i \in \mathcal{X}$ is an example and $y_i \in \mathcal{Y}$ is a label. Let $h : \mathcal{X} \to \mathcal{Y}$ be a hypothesized model (mapping from examples to labels) we are trying to evaluate. Let $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ be a non-negative *loss function* which measures how different two labels are.

The **empirical risk** is

$$R(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(x_i), y_i).$$

Most notions of error or accuracy in machine learning can be captured with an empirical risk. A simple example: measure the error rate on the dataset with the *0-1 Loss Function*

$$L(\hat{y}, y) = 0 \text{ if } \hat{y} = y \text{ and } 1 \text{ if } \hat{y} \neq y.$$

What are other examples you've seen of empirical risk and loss functions?

Loss function:

-
-
-
-
-
-

Regularizers:

- *

What are other examples you've seen of empirical risk and loss functions?

Loss function:

- Mean squared error
- Hinge loss
- Cross Entropy loss
- Absolute error
- Negative log likelihood
- ROC

Regularizers:

- ℓ_2 and ℓ_1 regularization

The Cost of Computing the Empirical Risk

$$R(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(x_i), y_i).$$

We need to compute the empirical risk a lot during training, both during validation (and hyperparameter optimization) and testing, so it's nice if we can do it fast.

Question: how does computing the empirical risk scale? Things affect the cost of computation. <!-- * The number of training examples n. The cost will certainly be proportional to n.

- The cost to compute the loss function L.
- The cost to evaluate the hypothesis h. -->

The Cost of Computing the Empirical Risk

$$R(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(x_i), y_i).$$

We need to compute the empirical risk a lot during training, both during validation (and hyperparameter optimization) and testing, so it's nice if we can do it fast.

Question: how does computing the empirical risk scale? Three things affect the cost of computation.

- The number of training examples n. The cost will certainly be proportional to n.
- The cost to compute the loss function L.
- The cost to evaluate the hypothesis h.

Review: Empirical Risk Minimization

We don't just want to compute the empirical risk: we also want to minimize it. To do so, we *parameterize* the hypotheses using some parameters $w \in \mathbb{R}^d$. That is, we assign each hypothesis a d-dimensional vector of parameters and vice versa and solve the optimization problem

$$\begin{aligned} \text{minimize: } & R(h_w) = \frac{1}{n} \sum_{i=1}^{n} L(h_w(x_i), y_i) = f(w) = \frac{1}{n} \sum_{x \in \mathcal{D}} f(w; x) \\ \text{over } & w \in \mathbb{R}^d \end{aligned}$$

where h_w denotes the hypothesis associated with the parameter vector w and $\mathcal{D}$ denotes the dataset. This is an instance of a principle of scalable ML:

Write your learning task as an optimization problem, then solve it with an optimization algorithm.

Gradient descent (GD).

Initialize the parameters at some value $w_0 \in \mathbb{R}^d$, and decrease the value of the empirical risk iteratively by running

$$w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t) = w_t - \alpha_t \cdot \frac{1}{n} \sum_{x \in \mathcal{D}} \nabla f(w; x)$$

where w_t is the value of the parameter vector at time t, α_t is a parameter called the *learning rate* or *step size*, and ∇f denotes the gradient (vector of partial derivatives) of f.

Gradient descent is not the only classic optimization method though...

Another iterative method of optimization: Newton's method.

Initialize the parameters at some value $w_0 \in \mathbb{R}^d$, and decrease the value of the empirical risk iteratively by running

$$w_{t+1} = w_t - \left(\nabla^2 f(w_t)\right)^{-1} \nabla f(w_t).$$

Newton's method can converge more quickly than gradient descent, because it is a *second-order optimization method* (it uses the second derivative of f) whereas gradient descent is only a *first-order method* (it uses the first derivative of f).

Question: what is the computational cost of running a single step of gradient descent and Newton's method? Not including the training set, how much memory is required?

Suppose that computing gradients of one example takes $\Theta(d)$ time and computing the Hessian of one example takes $\Theta(d^2)$ time, and express your answer in terms n and d.

Gradient descent: $w_{t+1} = w_t - \alpha_t \cdot \nabla f(w_t) = w_t - \alpha_t \cdot \frac{1}{n} \sum_{x \in \mathcal{D}} \nabla f(w; x)$ Time? Memory?

Time: $O(nd)$

Memory: $O(d)$

Newton's method: $w_{t+1} = w_t - \left(\nabla^2 f(w_t)\right)^{-1} \nabla f(w_t)$. Time? Memory?

Time: $O(nd^2 + d^3) = O(nd + nd^2 + d^3 + d^2 + d)$

Memory: $O(d^2)$

Why does gradient descent work?

...and what does it mean for it to work?

Intuitively, GD decreases the value of the objective at each iteration, as long as the learning rate is small enough and the value of the gradient is nonzero. Eventually, this should result in gradient descent *coming close to a point where the gradient is zero*. We can prove that gradient descent works under the assumption that *the second derivative of the objective is bounded* (this is sometimes called an L**-smoothness bound**).

- Why does this make sense?
  - The second derivative being bounded means that our evaluation of the gradient at w is a good approximation for the gradient nearby w
- There are other assumptions we could use here also, but this is the simplest one.

Lipschitz continuous gradient assumption (or the L-smoothness assumption)

Suppose that for some constant $L > 0$, for all x and y in $\mathbb{R}^d$,

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

Here, $\|\cdot\|$ denotes the ℓ_2 norm: $\|u\| = \sqrt{\sum_{i=1}^d u_i^2}$.

- It's saying the **the gradient can't change arbitrarily fast**
- This is equivalent to the condition that $\|\nabla^2 f(x)\|_2 \leq L$ (where the norm here denotes the induced norm) if the function is twice-differentiable.

Starting from this condition, let's look at how the objective changes over time as we run gradient descent with a fixed step size.

We will first prove the **descent lemma**, that is, for all x, y,

$$f(y) \leq f(x) + \langle \nabla f(x),\, y - x \rangle + \frac{L}{2}\|y - x\|^2$$

Consider the univariate function and its derivative

$$\begin{aligned} \rho(\tau) &= f(x + \tau(y - x)) && (1) \\ \rho'(\tau) &= \langle \nabla f(x + \tau(y - x)),\, y - x \rangle && (2) \end{aligned}$$

By the **Fundamental Theorem of Calculus**,

$$\rho(1) - \rho(0) = \int_0^1 \rho'(\tau)\, d\tau$$

If we substitute back in our definition of ρ, we get

$$f(y) - f(x) = \int_0^1 \langle \nabla f(x + \tau(y - x)),\, y - x >\ d\tau$$

$$f(y) = f(x) + \langle \nabla f(x),\, y - x \rangle + \int_0^1 \langle \nabla f(x + \tau(y - x)) - \nabla f(x),\, y - x >$$

where in the last step we rearranged, followed by an add and subtract of $\langle \nabla f(x),\, y - x \rangle$.

We can now simplify this as follows. Keep in mind: we assume that for all x, y, $\|\nabla f(x) - \nabla f(y)\| \leq L \|x - y\|$.

$$\begin{aligned} f(y) &= f(x) + \langle \nabla f(x), \, y - x \rangle + \int_0^1 \langle \nabla f(x + \tau(y-x)) - \nabla f(x), \, y - x > \, d\tau & (5) \\ &\leq f(x) + \langle \nabla f(x), \, y - x \rangle + \int_0^1 \|\nabla f(x + \tau(y-x)) - \nabla f(x)\| \, \|y - x\| \, d\tau & (6) \\ &\leq f(x) + \langle \nabla f(x), \, y - x \rangle + \int_0^1 L \, \|x + \tau(y-x) - x\| \, \|y - x\| \, d\tau & (7) \\ &= f(x) + \langle \nabla f(x), \, y - x \rangle + L\|y - x\|^2 \int_0^1 \tau \, d\tau & (8) \end{aligned}$$

where the first inequality comes from Cauchy-Schwarz inequality $a^\top b \leq \|a\| \, \|b\|$, and the second comes from our L-Lipschitz continuous gradient assumption. which gives us the **descent lemma**

$$f(y) \leq f(x) + \langle \nabla f(x), \, y - x \rangle + \frac{L}{2} \|y - x\|^2.$$

Descent lemma: Lipschitz conitnuous gradient implies a global quadratic upper bound

$$f(y) \leq f(x) + \langle \nabla f(x), \, y - x \rangle + \frac{L}{2} \|y - x\|^2.$$

(Figure shamelessly stolen from Mark Schmidt)

Why the descent lemma?

$$f(y) \leq f(x) + \langle \nabla f(x), \, y - x \rangle + \frac{L}{2} \|y - x\|^2.$$

Since it holds for all x, y, we can plug in w_t for x and w_{t+1} for y, and plug in the GD update $w_{t+1} = w_t - \alpha \nabla f(w_t)$,

$$\begin{aligned} f(w_{t+1}) &\leq f(w_t) + \langle \nabla f(w_t), \, w_{t+1} - w_t \rangle + \frac{L}{2} \|w_{t+1} - w_t\|^2 & (9) \\ &= f(w_t) - \alpha \langle \nabla f(w_t), \, \nabla f(w_t) \rangle + \frac{L}{2} \alpha^2 \|\nabla f(w_t)\|^2 & (10) \\ &= f(w_t) - \alpha \left(1 - \frac{1}{2} \alpha L\right) \|\nabla f(w_t)\|^2 & (11) \end{aligned}$$

Why the descent lemma?

So, we got to

$$f(w_{t+1}) \le f(w_t) - \alpha \left(1 - \frac{1}{2}\alpha L\right) \|\nabla f(w_t)\|^2 \tag{12}$$

If we choose our step size α to be smalle nough such that $\alpha L \le 1$, then $$f(w_{t+1}) \le f(w_t)

- \frac{\alpha}{2} \cdot | \nabla f(w_t) |^2$$ That is, **the objective is guaranteed to decrease at each iteration**! (Hence, **descent**)

This matches our intuition for why GD should work.

Convergence of GD

$$f(w_{t+1}) \le f(w_t) - \frac{\alpha}{2} \cdot \|\nabla f(w_t)\|^2$$

Now, if we sum this up across T iterations of gradient descent, we get

$$\frac{1}{2}\alpha \sum_{t=0}^{T-1} \|\nabla f(w_t)\|^2 \le \sum_{t=0}^{T-1} (f(w_t) - f(w_{t+1})) = f(w_0) - f(w_T) \le f(w_0) - f^*$$

where f^* is the global minimum value of the loss function f. From here, we can get

$$\min_{t \in \{0,...,T\}} \|\nabla f(w_t)\|^2 \le \frac{1}{T} \sum_{t=0}^{T-1} \|\nabla f(w_t)\|^2 \le \frac{2(f(w_0) - f^*)}{\alpha T}.$$

$$\min_{t \in \{0,...,T\}} \|\nabla f(w_t)\|^2 \le \frac{1}{T} \sum_{t=0}^{T-1} \|\nabla f(w_t)\|^2 \le \frac{2(f(w_0) - f^*)}{\alpha T}.$$

This means that the smallest gradient we observe after T iterations is getting smaller proportional to $1/T$. So gradient descent converges...

- as long as we look at the smallest observed gradient
- and we care about finding a point where the gradient is small

Question: does this ensure that gradient descent converges to the global optimum (i.e. the value of w that minimizes f over all $w \in \mathbb{R}^d$)?

...

Question: does this ensure that gradient descent converges to the global optimum (i.e. the value of w that minimizes f over all $w \in \mathbb{R}^d$)?

No, because f can have multiple local minima, and $\|\nabla f(w)\| = 0$ only implies a first-order stationary point. It can even be a local maximum or saddle point!

When can we ensure that gradient descent *does* converge to the unique global optimum?

Certainly, we can do this when there is only one global optimum.

The simplest case of this is the case of **convex objectives**.

Convex Functions

A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is convex if for all $x, y \in \mathbb{R}^d$, and all $\eta \in [0, 1]$

$$f(\eta x + (1 - \eta) y) \leq \eta f(x) + (1 - \eta) f(y).$$

What this means graphically is that if we draw a line segment between any two points in the graph of the function, that line segment will lie above the function. There are a bunch of equivalent conditions that work if the function is differentiable.

(Figure shamelessly stolen from Mark Schmidt)

Convex Functions

In terms of the gradient, for all $x, y \in \mathbb{R}^d$,

$$(x - y)^T \left(\nabla f(x) - \nabla f(y) \right) \geq 0,$$

and in terms of the Hessian, for all $x \in \mathbb{R}^d$ and all $u \in \mathbb{R}^d$

$$u^T \nabla^2 f(x) u \geq 0;$$

this is equivalent to saying that the Hessian is positive semidefinite.

Question: What are some examples of hypotheses and loss functions that result in a convex objective??

$$R(w) = \frac{1}{n} \sum_{i=1}^{n} L(h(w; x_i), y_i).$$

...

Question: What are some examples of hypotheses and loss functions that result in a convex objective??

$$R(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(x_i), y_i).$$

- Linear regression $R(w) = \frac{1}{n} \sum_{i=1}^{n} (w^T x_i - y_i)^2$
- Lasso $R(w) = \frac{1}{n} \sum_{i=1}^{n} |w^T x_i - y_i|$
- SVM $R(w) = \frac{1}{n} \sum_{i=1}^{n} \max\{0,\, 1 - y_i w^T x_i\} + \frac{\lambda}{2} \|w\|^2$
- ...

An even easier case than convexity: **strong convexity**.

A function is strongly convex with parameter $\mu > 0$ if for all $x, y \in \mathbb{R}^d$,

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2} \|x - y\|^2$$

A global quadratic *lower* bound!

(Figure shamelessly stolen from Mark Schmidt)

An even easier case than convexity: **strong convexity**.

or equivalently, for all $x \in \mathbb{R}^d$ and all $u \in \mathbb{R}^d$

$$u^T \nabla^2 f(x) u \geq \mu \|u\|^2.$$

One (somewhat weaker) condition that is implied by strong convexity is

$$\|\nabla f(x)\|^2 \geq 2\mu \left(f(x) - f^* \right);$$

this is sometimes called the *Polyak-Łojasiewicz condition*.

- A useful note: you can make any convex objective strongly convex by adding ℓ_2 regularization.

Gradient descent on strongly convex objectives.

As before, let's look at how the objective changes over time as we run gradient descent with a fixed step size.

- This is a standard approach when analyzing an iterative algorithm like gradient descent. From our **descent lemma** earlier, we had (as long as $1 \geq \alpha L$)

$$f(w_{t+1}) \leq f(w_t) - \frac{\alpha}{2} \cdot \|\nabla f(w_t)\|^2.$$

Now applying the Polyak-Lojasiewicz condition condition to this gives us

$$f(w_{t+1}) \le f(w_t) - \alpha\mu\left(f(w_t) - f^*\right).$$

Subtracting the global optimum f^* from both sides produces

$$f(w_{t+1}) - f^* \le f(w_t) - f^* - \alpha\mu\left(f(w_t) - f^*\right) = (1 - \alpha\mu)\left(f(w_t) - f^*\right).$$

So we got that

$$f(w_{t+1}) - f^* \le (1 - \alpha\mu)\left(f(w_t) - f^*\right).$$

But this means that

$$\begin{aligned} f(w_{t+2}) - f^* &\le (1 - \alpha\mu)\left(f(w_{t+1}) - f^*\right) \\ &\le (1 - \alpha\mu)\left((1 - \alpha\mu)\left(f(w_t) - f^*\right)\right) \\ &\le (1 - \alpha\mu)^2\left(f(w_t) - f^*\right). \end{aligned}$$

Applying this inductively, after K total steps

$$f(w_K) - f^* \le (1 - \alpha\mu)^K\left(f(w_0) - f^*\right) \le \exp(-\alpha\mu K)\cdot\left(f(w_0) - f^*\right).$$

This shows that, for strongly convex functions, **gradient descent with a constant step size converges exponentially quickly to the optimum**. This is sometimes called *convergence at a linear rate*.

If we use the largest step size that satisfies our earlier assumption that $1 \ge \alpha L$ (i.e. $\alpha = 1/L$), then this rate becomes

$$f(w_K) - f^* \le \exp\left(-\frac{\mu K}{L}\right)\cdot\left(f(w_0) - f^*\right).$$

Equivalently, in order to ensure that $f(w_K) - f^*$ for some target error $\epsilon > 0$, it suffices to set the number of iterations K large enough that

$$K \ge \frac{L}{\mu}\cdot\log\left(\frac{f(w_0) - f^*}{\epsilon}\right).$$

The main thing that affects how large K needs to be here is the ratio L/μ, which is a function just of the problem itself and not of how we initialize or how accurate we want our solutions to be. We call the ratio

$$\kappa = \frac{L}{\mu}$$

the **condition number of the problem**. The condition number encodes *how hard a strongly convex problem is to solve*.

- Observe that this ration is invariant to scaling of the objective function f.

When the condition number is very large, the problem can be very hard to solve, and take many iterations of gradient descent to get close to the optimum. Even if the cost of running gradient descent is tractible as we scale, **if scaling a problem causes the condition number to blow up, this can cause issues**!

What affects the condition number? What can we do to make the condition number smaller? DEMO.

In [2]:
```
import numpy
import scipy
import matplotlib
from matplotlib import pyplot
import http.client
matplotlib.rcParams['figure.figsize'] = (10, 7)
matplotlib.rcParams['font.size'] = 16
```

In [3]:
```
def process_libsvmdataset_line(line, num_features):
    (label_str, features_string) = line.split(" ",maxsplit=1)
    label = float(label_str)
    features = numpy.zeros(num_features)
    for f in features_string.split(" "):
        (i,v) = f.split(":");
        features[int(i)-1] = float(v);
    return (label,features)

def get_cpusmall_dataset():
    num_features = 12
    conn = http.client.HTTPSConnection("www.csie.ntu.edu.tw")
    conn.request("GET", "/~cjlin/libsvmtools/datasets/regression/cpusmall")
    r = conn.getresponse()
    assert(r.status == 200)
    cpusmall_dataset_bytes = r.read()
    cpusmall_dataset_str = cpusmall_dataset_bytes.decode("utf-8")
    cpusmall_dataset_lines = cpusmall_dataset_str.strip().split("\n")
    num_examples = len(cpusmall_dataset_lines)
    labels = numpy.zeros(num_examples)
    features = numpy.zeros((num_examples, num_features+1)) #add one extra featu
    for i in range(num_examples):
        (label,feats) = process_libsvmdataset_line(cpusmall_dataset_lines[i], n
        labels[i] = label
        features[i,0:num_features] = feats
        features[i,num_features] = 1.0
    return (labels, features)
```

In [23]:
```
# load the CPUsmall dataset from the libsvm datasets website
(cpusmall_labels, cpusmall_examples) = get_cpusmall_dataset();
```

This is a linear regression task, which has empirical risk

$$R(w) = \frac{1}{n} \sum_{i=1}^{n} (x_i^T w - y_i)^2$$

The gradient is

$$\nabla R(w) = \frac{2}{n} \sum_{i=1}^{n} (x_i^T w - y_i) x_i$$

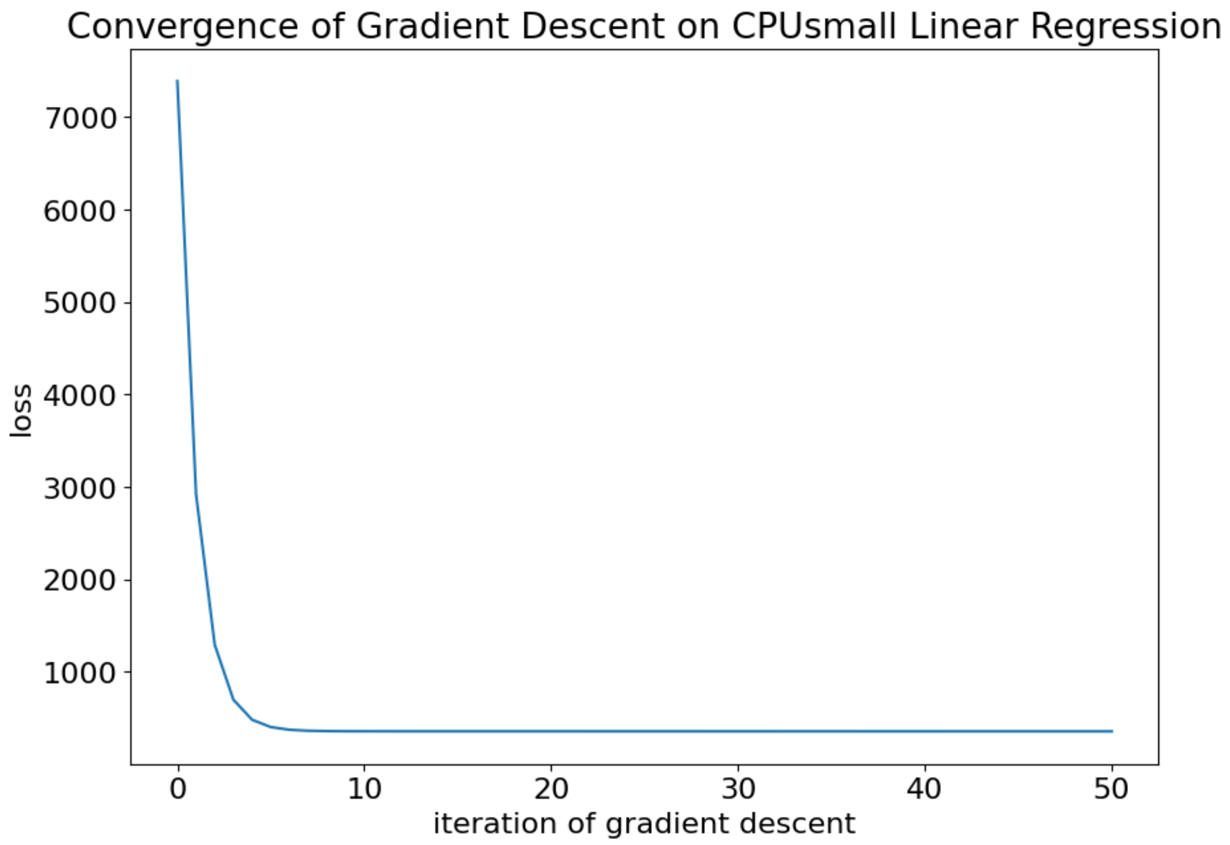
In [5]:
```
# loss of linear regression task
def linreg_loss(w, Xs, Ys):
    return numpy.mean((Xs @ w - Ys)**2)

# gradient of linear regression task
def linreg_grad(w, Xs, Ys):
    return (Xs.T @ (Xs @ w - Ys)) * (2/len(Ys))

def linreg_gradient_descent(Xs, Ys, alpha, w0, num_iters):
    w = w0;
    losses = numpy.zeros(num_iters + 1)
    for k in range(num_iters):
        losses[k] = linreg_loss(w, Xs, Ys)
        w = w - alpha * linreg_grad(w, Xs, Ys)
    losses[num_iters] = linreg_loss(w, Xs, Ys)
    return (w, losses);
```

In [6]:
```
# we can find the exact solution as follows
w_optimal = numpy.linalg.inv(cpusmall_examples.T @ cpusmall_examples) @ cpusma
loss_optimal = linreg_loss(w_optimal, cpusmall_examples, cpusmall_labels);

print(f"optimal loss: {loss_optimal}");
```

```
optimal loss: 96.36575268592969
```

In [7]:
```
w0 = numpy.zeros(13)
alpha = 1e-13 # setting the step size larger results in divergence!
num_iters = 50

(w_gd, losses_gd) = linreg_gradient_descent(cpusmall_examples, cpusmall_labels
```

In [8]:
```
%matplotlib inline

pyplot.plot(range(num_iters+1), losses_gd);
pyplot.xlabel("iteration of gradient descent");
pyplot.ylabel("loss");
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression");
```



Did gradient descent converge well on this example? Can we tell from this figure?

.

.

.

.

.

.

.

.

.

.

.

.

.

.

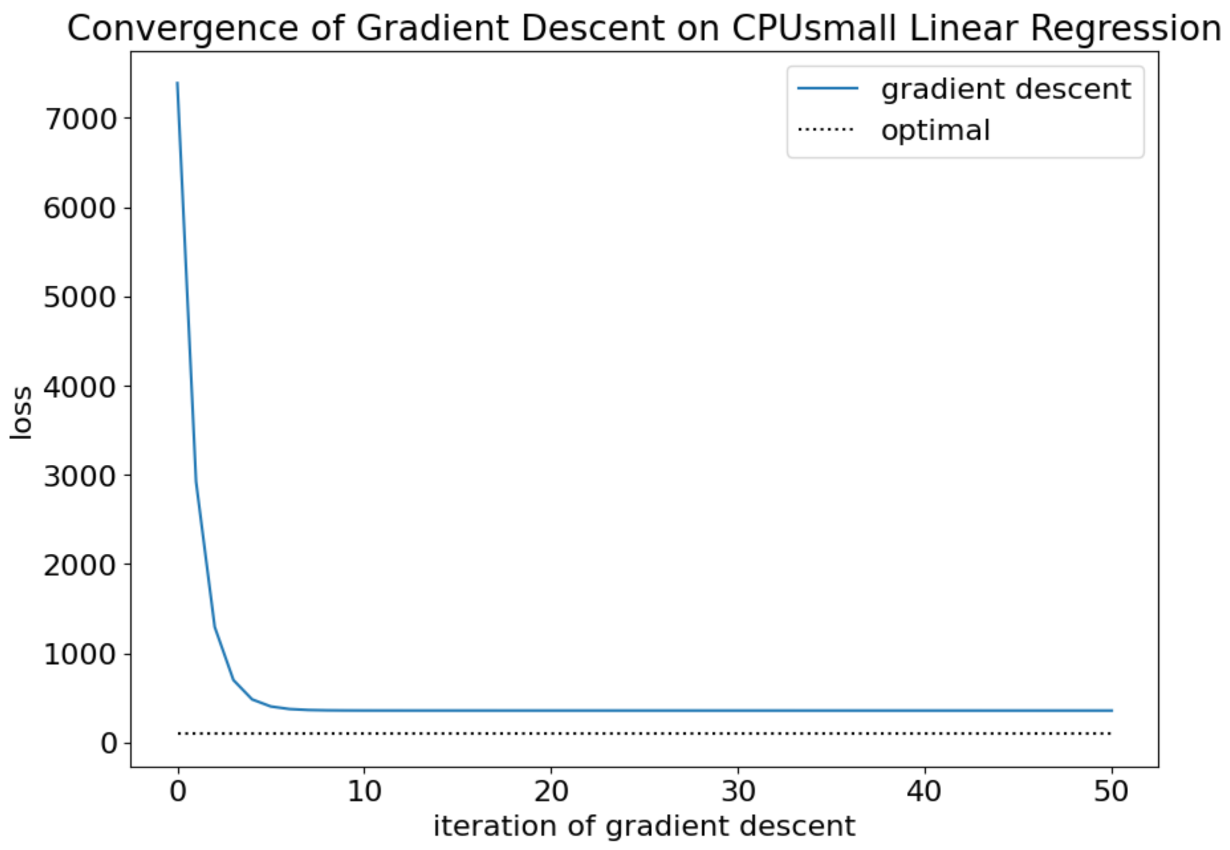
.

.

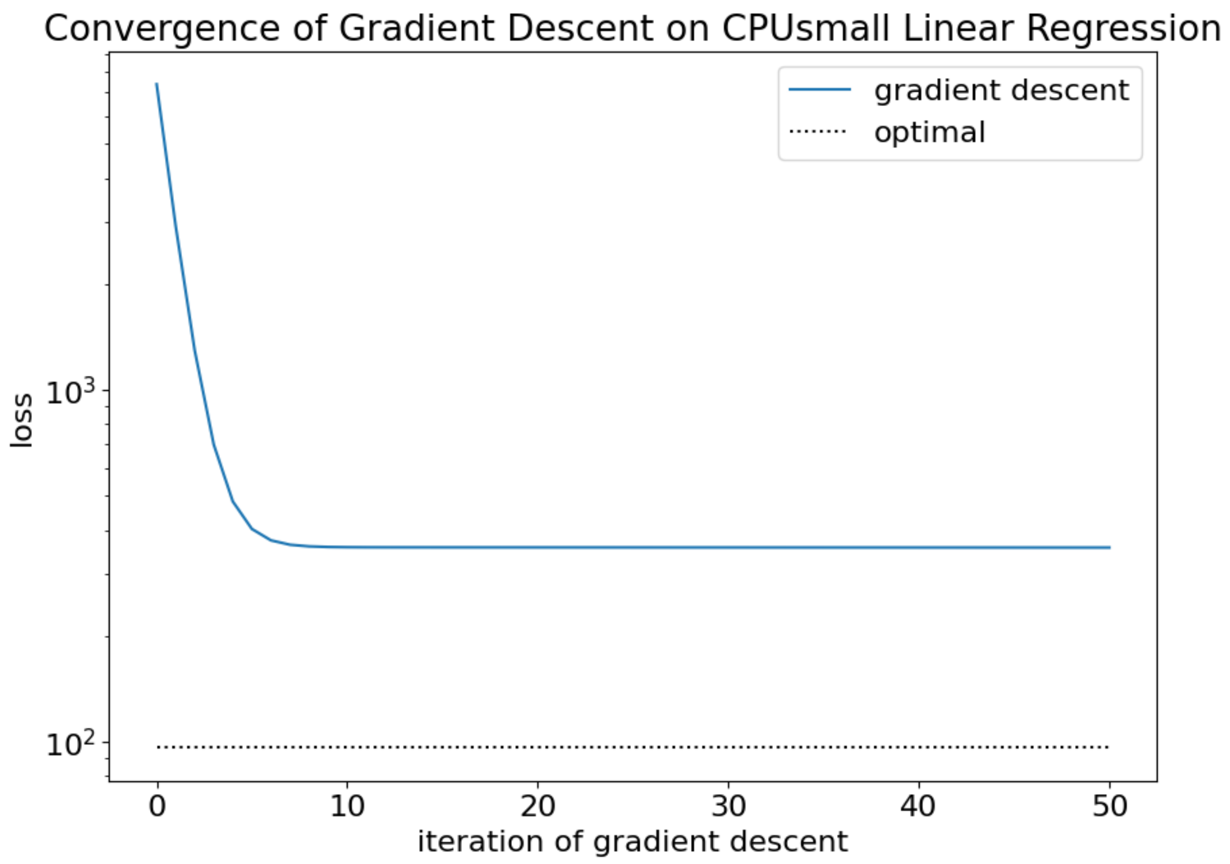
.

.

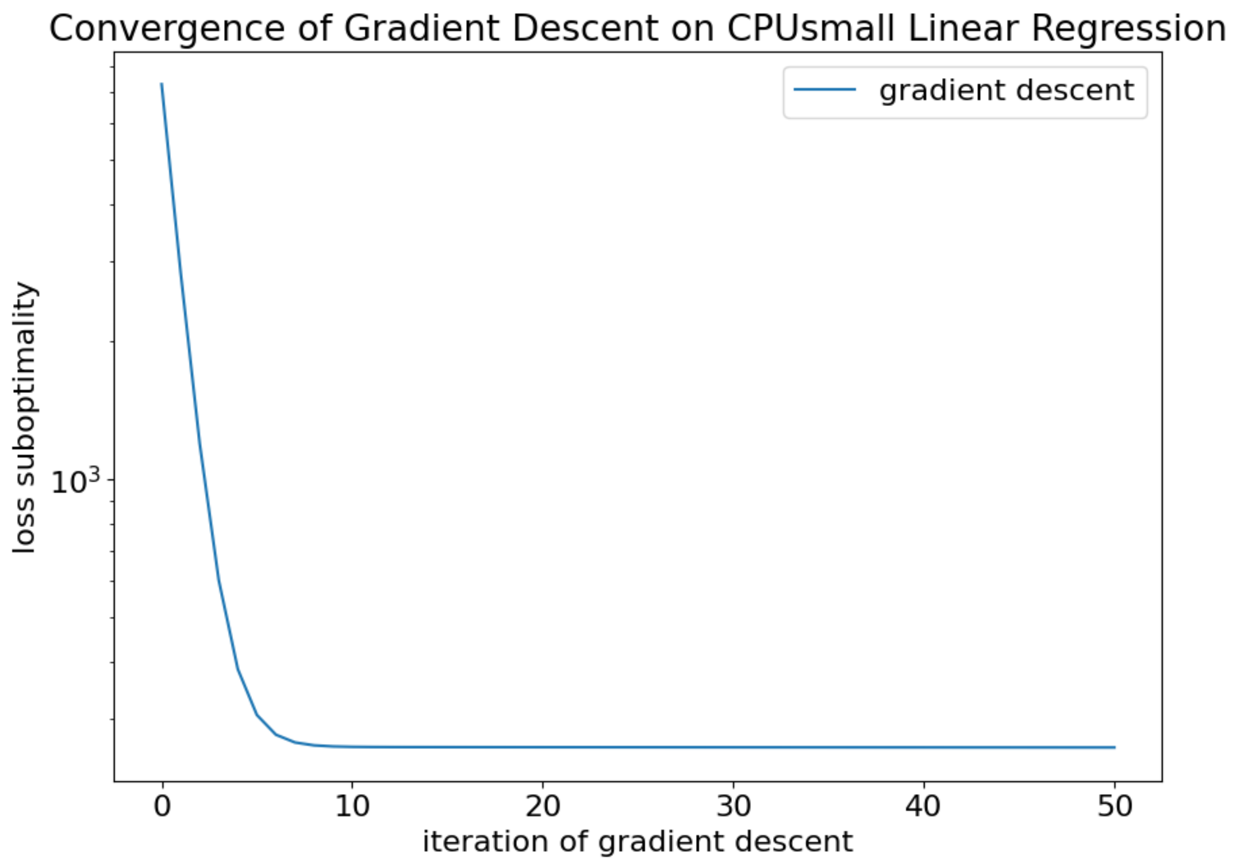
In [9]:
```
pyplot.plot(range(num_iters+1), losses_gd, label="gradient descent");
pyplot.plot(range(num_iters+1), 0 * losses_gd + loss_optimal, linestyle=":", c
pyplot.xlabel("iteration of gradient descent")
pyplot.ylabel("loss")
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression")
pyplot.legend();
```



In [10]:
```
pyplot.plot(range(num_iters+1), losses_gd, label="gradient descent");
pyplot.plot(range(num_iters+1), 0 * losses_gd + loss_optimal, linestyle=":", c
pyplot.xlabel("iteration of gradient descent")
pyplot.ylabel("loss")
pyplot.yscale("log")
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression")
pyplot.legend();
```



In [11]:
```
pyplot.plot(range(num_iters+1), losses_gd - loss_optimal, label="gradient desc
# pyplot.plot(range(num_iters+1), 0 * losses_gd + loss_optimal, linestyle=":",
pyplot.xlabel("iteration of gradient descent")
pyplot.ylabel("loss suboptimality")
pyplot.yscale("log")
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression")
pyplot.legend();
```



What went wrong? How could we fix this?

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Maybe looking at the dataset would help!

```
In [12]: cpusmall_examples[1,:]
```

```
Out[12]: array([1.000000e+00, 0.000000e+00, 2.165000e+03, 2.050000e+02,
                1.010000e+02, 4.000000e-01, 1.200000e+00, 4.310700e+04,
                4.413900e+04, 3.000000e+00, 1.300000e+02, 1.131931e+06,
                1.000000e+00])
```

Does this give any new insights?

.

.

.

.

.

.

.

.

.

.

.

.

.

.

```
In [13]: # normalize the examples in two ways
```

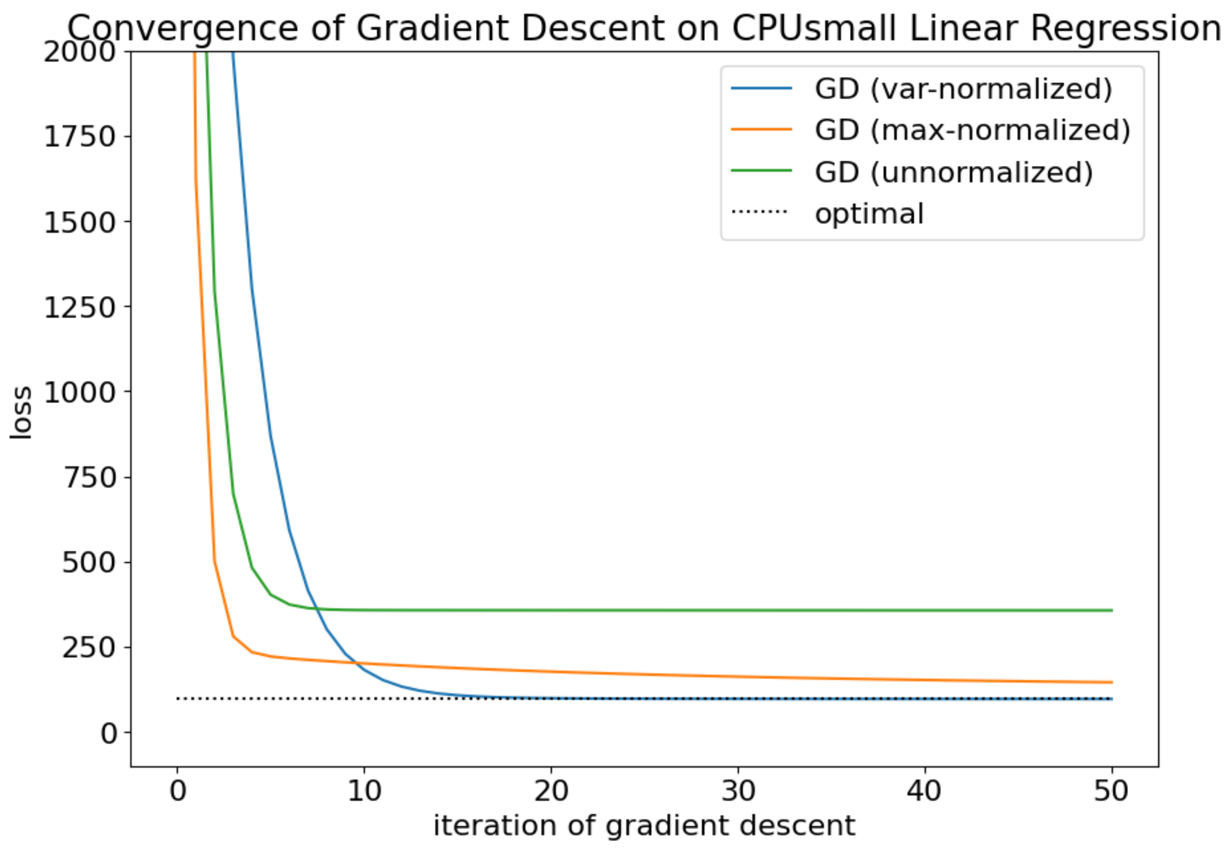
```
# mean = 0, variance = 1
cpusmall_mean = numpy.mean(cpusmall_examples, axis=0)
cpusmall_mean[12] = 0.0
cpusmall_var = numpy.mean((cpusmall_examples - cpusmall_mean)**2, axis=0)
cpusmall_normalized1 = (cpusmall_examples - cpusmall_mean) / numpy.sqrt(cpusma

# scale by max all features lie in [-1,1]
cpusmall_max = numpy.max(numpy.abs(cpusmall_examples), axis=0)
cpusmall_normalized2 = cpusmall_examples / cpusmall_max
```

In [14]:
```
# we can find the exact solution as follows
w_opt_normalized1 = numpy.linalg.inv(cpusmall_normalized1.T @ cpusmall_normali
loss_opt_normalized1 = linreg_loss(w_opt_normalized1, cpusmall_normalized1, cpu

w_opt_normalized2 = numpy.linalg.inv(cpusmall_normalized2.T @ cpusmall_normali
loss_opt_normalized2 = linreg_loss(w_opt_normalized2, cpusmall_normalized2, cpu

# should all be the same
print(f"optimal loss (without normalization): {loss_optimal}");
print(f"optimal loss (with  normalization 1): {loss_opt_normalized1}");
print(f"optimal loss (with  normalization 2): {loss_opt_normalized2}");
```

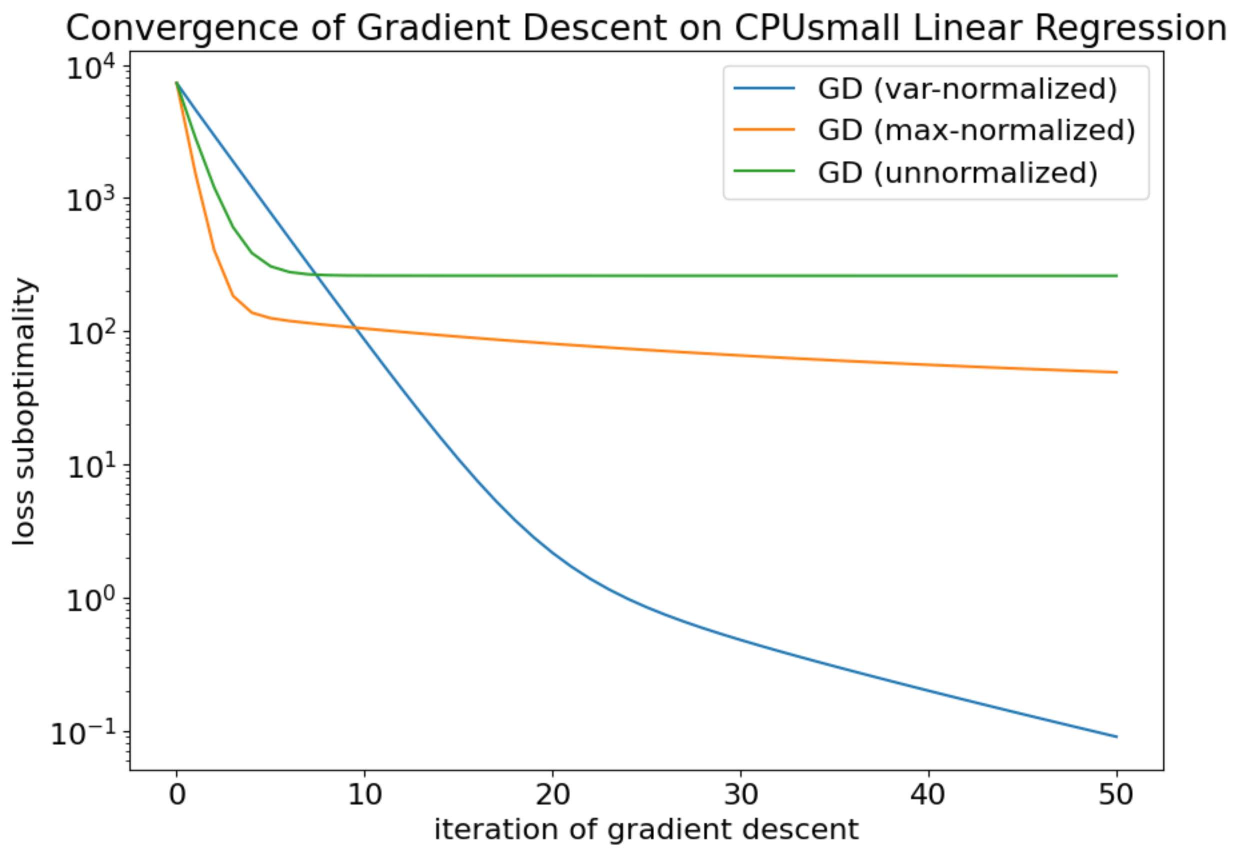
```
optimal loss (without normalization): 96.36575268592969
optimal loss (with  normalization 1): 96.36575268592969
optimal loss (with  normalization 2): 96.3657526859297
```

In [15]:
```
w0 = numpy.zeros(13)
alpha = 0.1 # setting the step size larger results in divergence!
(w_gdn, losses_gdn1) = linreg_gradient_descent(cpusmall_normalized1, cpusmall_

w0 = numpy.zeros(13)
alpha = 0.5 # setting the step size larger results in divergence!
(w_gdn, losses_gdn2) = linreg_gradient_descent(cpusmall_normalized2, cpusmall_
```

In [16]:
```
pyplot.plot(range(num_iters+1), losses_gdn1, label="GD (var-normalized)");
pyplot.plot(range(num_iters+1), losses_gdn2, label="GD (max-normalized)");
pyplot.plot(range(num_iters+1), losses_gd, label="GD (unnormalized)");
pyplot.plot(range(num_iters+1), 0 * losses_gd + loss_optimal, linestyle=":", c
pyplot.xlabel("iteration of gradient descent");
pyplot.ylabel("loss");
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression");
pyplot.ylim((-100,2000))
pyplot.legend();
```



In [17]:
```
pyplot.plot(range(num_iters+1), losses_gdn1-loss_optimal, label="GD (var-norma
pyplot.plot(range(num_iters+1), losses_gdn2-loss_optimal, label="GD (max-norma
pyplot.plot(range(num_iters+1), losses_gd-loss_optimal, label="GD (unnormalize
# pyplot.plot(range(num_iters+1), 0 * losses_gd + loss_optimal, linestyle=":",
pyplot.xlabel("iteration of gradient descent");
pyplot.ylabel("loss suboptimality");
pyplot.yscale("log");
pyplot.title("Convergence of Gradient Descent on CPUsmall Linear Regression");
# pyplot.ylim((-100,2000))
pyplot.legend();
```



Can we interpret this in terms of the condition number?

For linear regression,

$$\nabla^2 R(w) = 2 \frac{1}{n} \sum_{i=1}^{n} x_i x_i^T.$$

In [18]:
```
D2R_unnormalized = (cpusmall_examples.T @ cpusmall_examples) * 2 / len(cpusmal
D2R_normalized1 = (cpusmall_normalized1.T @ cpusmall_normalized1) * 2 / len(cpu
D2R_normalized2 = (cpusmall_normalized2.T @ cpusmall_normalized2) * 2 / len(cpu
```

In [19]:
```
def condition_number(H):
    # make sure H is symmetric for the eigendecomposition
    H = (H + H.T) / 2;
    ev = numpy.linalg.eigvals(H);
    mu = numpy.min(ev);
    L = numpy.max(ev);
    kappa = L / mu;
    return kappa;
```

In [20]:
```
print(f"unnormalized, kappa = {(condition_number(D2R_unnormalized))}")
```
```
unnormalized, kappa = 49943068264104.69
```

In [21]:
```
print(f"var normalized, kappa = {(condition_number(D2R_normalized1))}")
```
```
var normalized, kappa = 43.805303354607176
```

In [22]:
```
print(f"max normalized, kappa = {(condition_number(D2R_normalized2))}")
```

```
max normalized, kappa = 14128.211688196923
```

What can we conclude from this demo?

...

What can we conclude from this demo?

Large condition number, slower convergence!

Problem Set 2 is out on Gradescope!