CSC 311: Introduction to Machine Learning

Lecture 4 - Linear Classification & Optimization

Richard Zemel & Murat A. Erdogdu

University of Toronto

Overview

- Classification: predicting a discrete-valued target
  - Binary classification: predicting a binary-valued target
- Examples
  - predict whether a patient has a disease, given the presence or absence of various symptoms
  - classify e-mails as spam or non-spam
  - predict whether a financial transaction is fraudulent

Overview

Binary linear classification

- **classification:** predict a discrete-valued target
- **binary:** predict a binary target $t \in \{0, 1\}$
  - Training examples with $t = 1$ are called positive examples, and training examples with $t = 0$ are called negative examples. Sorry.
  - $t \in \{0, 1\}$ or $t \in \{-1, +1\}$ is for computational convenience.
- **linear:** model is a linear function of $\mathbf{x}$, followed by a threshold r:

$$z = \mathbf{w}^T\mathbf{x} + b$$

$$y = \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases}$$

Some simplifications

Eliminating the threshold

- We can assume WLOG that the threshold $r = 0$:

$$\mathbf{w}^T\mathbf{x} + b \geq r \quad \Longleftrightarrow \quad \mathbf{w}^T\mathbf{x} + \underbrace{b - r}_{\triangleq w_0} \geq 0.$$

Eliminating the bias

- Add a dummy feature x_0 which always takes the value 1. The weight $w_0 = b$ is equivalent to a bias (same as linear regression)

Simplified model

$$z = \mathbf{w}^T\mathbf{x}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Examples

- Let's consider some simple examples to examine the properties of our model
- Forget about generalization and suppose we just want to learn Boolean functions

Examples

NOT

x_0	x_1	t
1	0	1
1	1	0

- This is our "training set"
- What conditions are needed on w_0, w_1 to classify all examples?
  - When $x_1 = 0$, need: $z = w_0 x_0 + w_1 x_1 > 0 \iff w_0 > 0$
  - When $x_1 = 1$, need: $z = w_0 x_0 + w_1 x_1 < 0 \iff w_0 + w_1 < 0$
- Example solution: $w_0 = 1, w_1 = -2$
- Is this the only solution?

Examples

AND

x_0	x_1	x_2	t
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

$$z = w_0x_0 + w_1x_1 + w_2x_2$$

need: $w_0 < 0$

need: $w_0 + w_2 < 0$

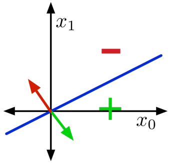
need: $w_0 + w_1 < 0$

need: $w_0 + w_1 + w_2 > 0$

Example solution: $w_0 = -1.5$, $w_1 = 1$, $w_2 = 1$

The Geometric Picture

Input Space, or Data Space for NOT example

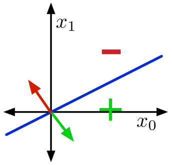


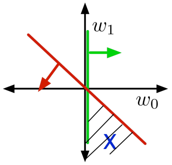
x_0	x_1	t
1	0	1
1	1	0

- Training examples are points
- Weights (hypotheses) $\mathbf{w}$ can be represented by half-spaces $H_+ = \{\mathbf{x} : \mathbf{w}^T\mathbf{x} \geq 0\}$, $H_- = \{\mathbf{x} : \mathbf{w}^T\mathbf{x} < 0\}$
  - The boundaries of these half-spaces pass through the origin (why?)
- The boundary is the decision boundary: $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} = 0\}$
  - In 2-D, it's a line, but think of it as a hyperplane
- If the training examples can be perfectly separated by a linear decision rule, we say data is linearly separable.

The Geometric Picture

Weight Space





$w_0 > 0$

$w_0 + w_1 < 0$

- Weights (hypotheses) $\mathbf{w}$ are points
- Each training example $\mathbf{x}$ specifies a half-space $\mathbf{w}$ must lie in to be correctly classified: $\mathbf{w}^T\mathbf{x} > 0$ if $t = 1$.
- For NOT example:
  - $x_0 = 1, x_1 = 0, t = 1 \implies (w_0, w_1) \in \{\mathbf{w} : w_0 > 0\}$
  - $x_0 = 1, x_1 = 1, t = 0 \implies (w_0, w_1) \in \{\mathbf{w} : w_0 + w_1 < 0\}$
- The region satisfying all the constraints is the feasible region; if this region is nonempty, the problem is feasible, otw it is infeasible.

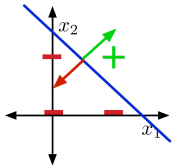
The Geometric Picture

- The **AND** example requires three dimensions, including the dummy one.
- To visualize data space and weight space for a 3-D example, we can look at a 2-D slice.
- The visualizations are similar.
  - Feasible set will always have a corner at the origin.

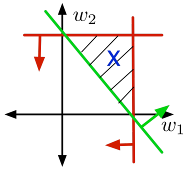
The Geometric Picture

Visualizations of the **AND** example

Data Space



Weight Space



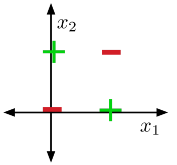
- Slice for $x_0 = 1$ and
- example sol: $w_0 = -1.5$, $w_1 = 1$, $w_2 = 1$
- decision boundary:

$w_0x_0 + w_1x_1 + w_2x_2 = 0$
$\implies -1.5 + x_1 + x_2 = 0$

- Slice for $w_0 = -1.5$ for the constraints
- $w_0 < 0$
- $w_0 + w_2 < 0$
- $w_0 + w_1 < 0$
- $w_0 + w_1 + w_2 > 0$

The Geometric Picture

Some datasets are not linearly separable, e.g. **XOR**



Overview

- **Recall: binary linear classifiers.** Targets $t \in \{0, 1\}$

$$z = \mathbf{w}^T\mathbf{x} + b$$
$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

- How can we find good values for $\mathbf{w}, b$?
- If training set is separable, we can solve for $\mathbf{w}, b$ using linear programming
- If it's not separable, the problem is harder
  - data is almost never separable in real life.

Loss functions

- Instead: define loss function then try to minimize the resulting cost function
  - Recall: cost is loss averaged (or summed) over the training set
- Seemingly obvious loss function: 0-1 loss

$$\begin{aligned}\mathcal{L}_{0-1}(y,t) &= \begin{cases} 0 & \text{if } y = t \\ 1 & \text{if } y \neq t \end{cases} \\ &= \mathbb{I}[y \neq t]\end{aligned}$$

Attempt 1: 0-1 loss

- Usually, the cost $\mathcal{J}$ is the averaged loss over training examples; for 0-1 loss, this is the misclassification rate:

$$\mathcal{J} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}[y^{(i)} \neq t^{(i)}]$$

Attempt 1: 0-1 loss

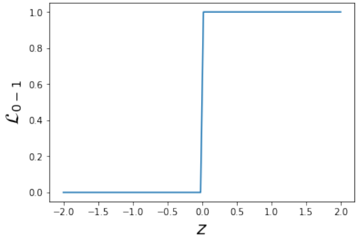
- Problem: how to optimize? In general, a hard problem (can be NP-hard)
- This is due to the step function (0-1 loss) not being nice (continuous/smooth/convex etc)

Attempt 1: 0-1 loss

- Minimum of a function will be at its critical points.
- Let's try to find the critical point of 0-1 loss
- Chain rule:

$$\frac{\partial \mathcal{L}_{0-1}}{\partial w_j} = \frac{\partial \mathcal{L}_{0-1}}{\partial z} \frac{\partial z}{\partial w_j}$$

- But $\partial \mathcal{L}_{0-1} / \partial z$ is zero everywhere it's defined!



  - $\partial \mathcal{L}_{0-1} / \partial w_j = 0$ means that changing the weights by a very small amount probably has no effect on the loss.
  - Almost any point has 0 gradient!

Attempt 2: Linear Regression

- Sometimes we can replace the loss function we care about with one which is easier to optimize. This is known as relaxation with a smooth surrogate loss function.
- One problem with $\mathcal{L}_{0-1}$: defined in terms of final prediction, which inherently involves a discontinuity
- Instead, define loss in terms of $\mathbf{w}^T\mathbf{x} + b$ directly
  - Redo notation for convenience: $z = \mathbf{w}^T\mathbf{x} + b$

Attempt 2: Linear Regression

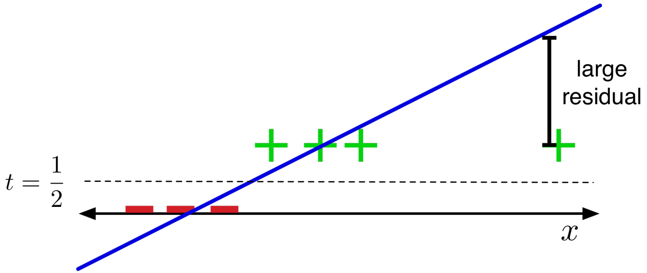
- We already know how to fit a linear regression model. Can we use this instead?

$$z = \mathbf{w}^\top \mathbf{x} + b$$
$$\mathcal{L}_{\mathrm{SE}}(z, t) = \frac{1}{2}(z - t)^2$$

- Doesn't matter that the targets are actually binary. Treat them as continuous values.
- For this loss function, it makes sense to make final predictions by thresholding z at $\frac{1}{2}$ (why?)

Attempt 2: Linear Regression

The problem:



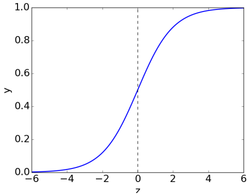
- The loss function hates when you make correct predictions with high confidence!
- If $t = 1$, it's more unhappy about $z = 10$ than $z = 0$.

Attempt 3: Logistic Activation Function

- There's obviously no reason to predict values outside [0, 1]. Let's squash y into this interval.

- The logistic function is a kind of sigmoid, or S-shaped function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



- $\sigma^{-1}(y) = \log(y/(1-y))$ is called the logit.
- A linear model with a logistic nonlinearity is known as log-linear:

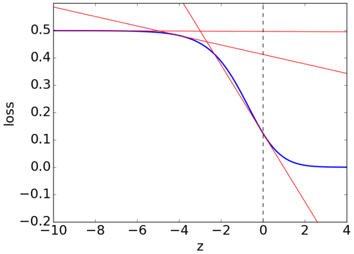
$$z = \mathbf{w}^\top \mathbf{x} + b$$
$$y = \sigma(z)$$
$$\mathcal{L}_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2.$$

- Used in this way, σ is called an activation function.

Attempt 3: Logistic Activation Function

The problem:

(plot of $\mathcal{L}_{\text{SE}}$ as a function of z, assuming $t = 1$)



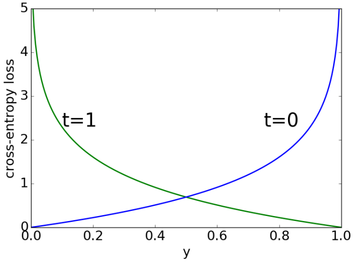
$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w_j}$$

- For $z \ll 0$, we have $\sigma(z) \approx 0$.
- $\frac{\partial \mathcal{L}}{\partial z} \approx 0$ (check!) $\implies$ $\frac{\partial \mathcal{L}}{\partial w_j} \approx 0$ $\implies$ derivative w.r.t. w_j is small $\implies$ w_j is like a critical point
- If the prediction is really wrong, you should be far from a critical point (which is your candidate solution).

Logistic Regression

- Because $y \in [0,1]$, we can interpret it as the estimated probability that $t = 1$.
- The pundits who were 99% confident Clinton would win were much more wrong than the ones who were only 90% confident.
- Cross-entropy loss (aka log loss) captures this intuition:

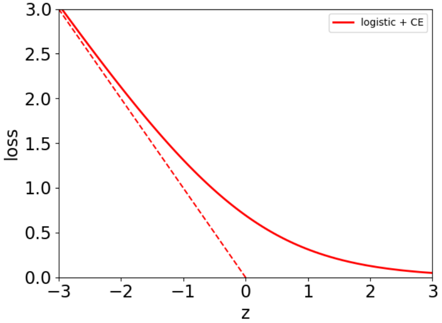
$$\mathcal{L}_{\mathrm{CE}}(y,t) = \begin{cases} -\log y & \text{if } t = 1 \\ -\log(1-y) & \text{if } t = 0 \end{cases}$$
$$= -t \log y - (1-t)\log(1-y)$$



Logistic Regression

Logistic Regression:

$$\begin{aligned} z &= \mathbf{w}^\top \mathbf{x} + b \\ y &= \sigma(z) \\ &= \frac{1}{1+e^{-z}} \\ \mathcal{L}_{\text{CE}} &= -t \log y - (1-t)\log(1-y) \end{aligned}$$



Plot is for target $t = 1$.

Logistic Regression

- Problem: what if $t = 1$ but you're really confident it's a negative example ($z \ll 0$)?
- If y is small enough, it may be numerically zero. This can cause very subtle and hard-to-find bugs.

$$\begin{aligned} y &= \sigma(z) && \Rightarrow y \approx 0 \\ \mathcal{L}_{\text{CE}} &= -t \log y - (1-t)\log(1-y) && \Rightarrow \text{ computes } \log 0 \end{aligned}$$

- Instead, we combine the activation function and the loss into a single logistic-cross-entropy function.

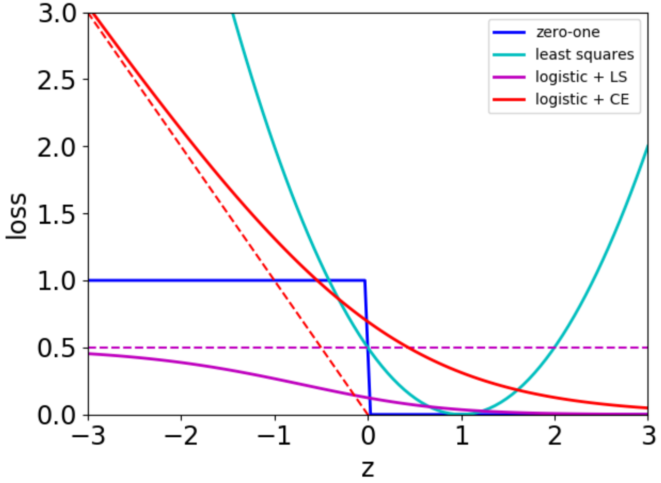
$$\mathcal{L}_{\text{LCE}}(z,t) = \mathcal{L}_{\text{CE}}(\sigma(z),t) = t\log(1+e^{-z}) + (1-t)\log(1+e^{z})$$

- Numerically stable computation:

```
E = t * np.logaddexp(0, -z) + (1-t) * np.logaddexp(0, z)
```

Logistic Regression

Comparison of loss functions: (for $t = 1$)



Gradient Descent

- How do we minimize the cost $\mathcal{J}$ in this case? No direct solution.
  - Taking derivatives of $\mathcal{J}$ w.r.t. $\mathbf{w}$ and setting them to 0 doesn't have an explicit solution.
- Now let's see a second way to minimize the cost function which is more broadly applicable: gradient descent.
- Gradient descent is an iterative algorithm, which means we apply an update repeatedly until some criterion is met.
- We initialize the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the direction of steepest descent.

Gradient descent

- This is an iterative algorithm to minimize a cost function $\mathcal{J}(\mathbf{w})$
- It uses the update rule in vector form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

- This gets its name from the gradient:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

  - This is the direction of fastest increase in $\mathcal{J}$.
- $\alpha \in (0, 1]$ is the learning rate (or step size). More on this soon.
- Hence, gradient descent updates the weights in the direction of fastest *decrease*.
- Observe that once it converges, we get a critical point, i.e. $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = 0$.

Gradient descent under L^2 Regularization

- Gradient descent update to minimize $\mathcal{J}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} \mathcal{J}$$

- The gradient descent update to minimize the regularized cost $\mathcal{J} + \lambda \mathcal{R}$ results in weight decay:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} \left(\mathcal{J} + \lambda \mathcal{R}\right) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\
&= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\
&= (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}
\end{aligned}$$

Descent on a coordinate

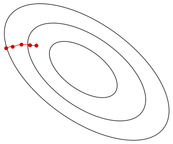
- Observe:
  - if $\partial \mathcal{J}/\partial w_j > 0$, then increasing w_j increases $\mathcal{J}$.
  - if $\partial \mathcal{J}/\partial w_j < 0$, then increasing w_j decreases $\mathcal{J}$.
- The following update always decreases the cost function for small enough α (unless $\partial \mathcal{J}/\partial w_j = 0$):

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

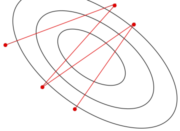
- $\alpha \in (0, 1]$ is a learning rate (or step size). The larger it is, the faster $\mathbf{w}$ changes.
  - We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001.
  - If cost is the sum of N individual losses rather than their average, smaller learning rate will be needed ($\alpha' = \alpha/N$).

Learning Rate (step size)

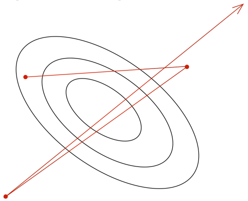
- In gradient descent, the learning rate α is a hyperparameter we need to tune. Here are some things that can go wrong:



α too small:
slow progress



α too large:
oscillations

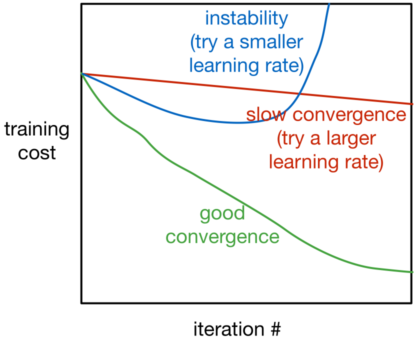


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try $0.1, 0.03, 0.01, \ldots$).

Training Curves

- To diagnose optimization problems, it's useful to look at training curves: plot the training cost as a function of iteration.



- Warning: it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.

Gradient of logistic loss

Back to logistic regression:

$$\begin{aligned}\mathcal{L}_{\text{CE}}(y,t) =& -t\log(y) - (1-t)\log(1-y)\\ y =& 1/(1+e^{-z}) \quad \text{and} \quad z = \mathbf{w}^T\mathbf{x} + b\end{aligned}$$

Therefore

$$\begin{aligned}\frac{\partial \mathcal{L}_{\text{CE}}}{\partial w_j} = \frac{\partial \mathcal{L}_{\text{CE}}}{\partial y}\cdot\frac{\partial y}{\partial z}\cdot\frac{\partial z}{\partial w_j} =& \left(-\frac{t}{y} + \frac{1-t}{1-y}\right)\cdot y(1-y)\cdot x_j\\ =& (y-t)x_j\end{aligned}$$

Gradient descent (coordinatewise) update to find the weights of logistic regression:

$$\begin{aligned}w_j &\leftarrow w_j - \alpha\frac{\partial \mathcal{J}}{\partial w_j}\\ &= w_j - \frac{\alpha}{N}\sum_{i=1}^{N}(y^{(i)} - t^{(i)})\,x_j^{(i)}\end{aligned}$$

Gradient descent for Linear regression

- Even for linear regression, where there is a direct solution, we sometimes need to use GD.
- Why gradient descent, if we can find the optimum directly?
  - GD can be applied to a much broader set of models
  - GD can be easier to implement than direct solutions
  - For regression in high-dimensional spaces, GD is more efficient than direct solution
    - Linear regression solution: $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t}$
    - matrix inversion is an $\mathcal{O}(D^3)$ algorithm
    - each GD update costs $O(ND)$
    - Huge difference if $D \gg 1$

Logistic Regression

Comparison of gradient descent updates:

- Linear regression (verify!):

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, \mathbf{x}^{(i)}$$

- Logistic regression:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, \mathbf{x}^{(i)}$$

- Not a coincidence! These are both examples of generalized linear models. But we won't go in further detail.
- Notice $\frac{1}{N}$ in front of sums due to averaged losses. This is why you need smaller learning rate when cost is summed losses ($\alpha' = \alpha/N$).

Stochastic Gradient Descent

- So far, the cost function $\mathcal{J}$ has been the average loss over the training examples:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}^{(i)} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

- By linearity,

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}.$$

- Computing the gradient requires summing over *all* of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset $N \gg 1$ (e.g. millions of training examples)!

Stochastic Gradient Descent

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example,

$$1 - \text{Choose } i \text{ uniformly at random,} \quad 2 - \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$$

- Cost of each SGD update is independent of N!
- SGD can make significant progress before even seeing all the data!
- Mathematical justification: if you sample a training example uniformly at random, the stochastic gradient is an unbiased estimate of the batch gradient:

$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}\right] = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}.$$

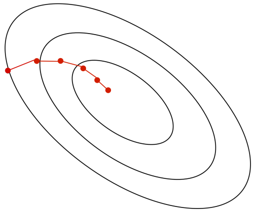
- Problems:
  - Variance in this estimate may be high
  - If we only look at one training example at a time, we can't exploit efficient vectorized operations.

Stochastic Gradient Descent

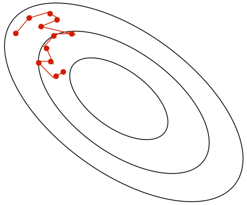
- Compromise approach: compute the gradients on a randomly chosen medium-sized set of training examples $\mathcal{M} \subset \{1, \ldots, N\}$, called a mini-batch.
- Stochastic gradients computed on larger mini-batches have smaller variance. This is similar to bagging.
- The mini-batch size $|\mathcal{M}|$ is a hyperparameter that needs to be set.
  - Too large: takes more computation, i.e. takes more memory to store the activations, and longer to compute each gradient update
  - Too small: can't exploit vectorization, has high variance
  - A reasonable value might be $|\mathcal{M}| = 100$.

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill. SGD takes steps in a noisy direction, but moves downhill on average.



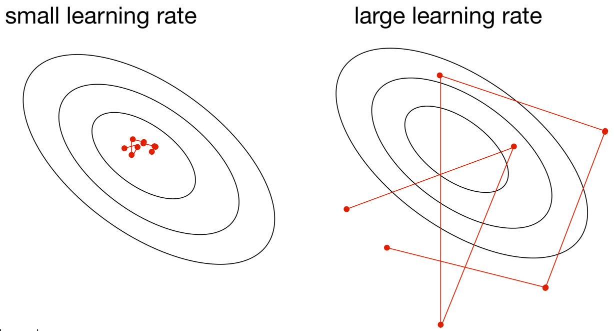
batch gradient descent



stochastic gradient descent

SGD Learning Rate

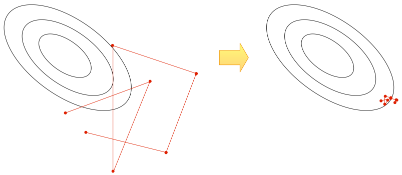
- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

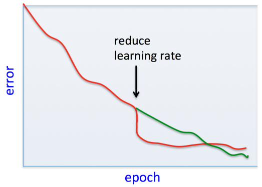


- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

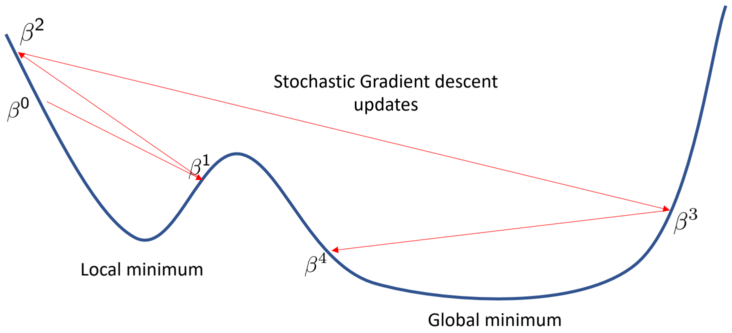
SGD Learning Rate

- Warning: by reducing the learning rate, you reduce the fluctuations, which can appear to make the loss drop suddenly. But this can come at the expense of long-run performance.





SGD and Non-convex optimization



- Stochastic methods have a chance of escaping from bad minima.
- Gradient descent with small step-size converges to first minimum it finds.

Conclusion

- We talked about linear methods for binary classification.
- We learned a non-linear model: logistic regression
  - but had no direct solution!
- We learned gradient descent, a method to minimize general cost functions.
- We learned stochastic gradient descent which is the most common technique used to train ML algorithms.