

CSC321 Lecture 4: Learning a Classifier

Roger Grosse

Overview

- Last time: binary classification, perceptron algorithm
- Limitations of the perceptron
 - no guarantees if data aren't linearly separable
 - how to generalize to multiple classes?
 - linear model — no obvious generalization to multilayer neural networks
- This lecture: apply the strategy we used for linear regression
 - define a model and a cost function
 - optimize it using gradient descent

Overview

Design choices so far

- **Task:** regression, binary classification, multiway classification
- **Model/Architecture:** linear, log-linear
- **Loss function:** squared error, 0–1 loss, cross-entropy, hinge loss
- **Optimization algorithm:** direct solution, gradient descent, perceptron

Overview

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

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

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

- Goal from last lecture: classify all training examples correctly
 - But what if we can't, or don't want to?
- Seemingly obvious loss function: **0-1 loss**

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

Attempt 1: 0-1 loss

- As always, the cost \mathcal{E} is the average loss over training examples; for 0-1 loss, this is the **error rate**:

$$\mathcal{E} = \frac{1}{N} \sum_{i=1}^N \mathbb{1}_{y^{(i)} \neq t^{(i)}}$$

$$\frac{1}{3} \left(\begin{array}{|c|} \hline \blacksquare \\ \hline \end{array} + \begin{array}{|c|} \hline \square \\ \hline \end{array} + \begin{array}{|c|} \hline \square \diagdown \blacksquare \\ \hline \end{array} \right) = \begin{array}{|c|} \hline \begin{array}{cc} \square & \square \\ \hline \blacksquare & \blacksquare \end{array} \\ \hline \end{array}$$

Attempt 1: 0-1 loss

- Problem: how to optimize?
- Chain rule:

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

Attempt 1: 0-1 loss

- Problem: how to optimize?
- 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.
 - The gradient descent update is a no-op.

Attempt 2: Linear Regression

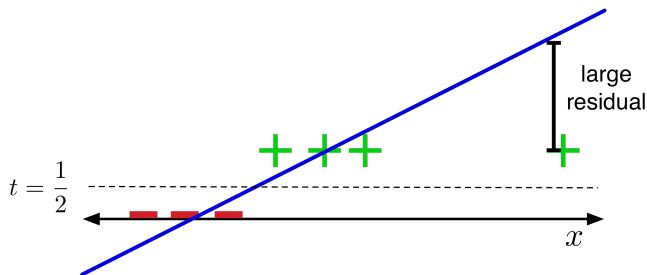
- Sometimes we can replace the loss function we care about with one which is easier to optimize. This is known as a **surrogate loss function**.
- We already know how to fit a linear regression model. Can we use this instead?

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

- Doesn't matter that the targets are actually binary.
- Threshold predictions at $y = 1/2$.

Attempt 2: Linear Regression

The problem:

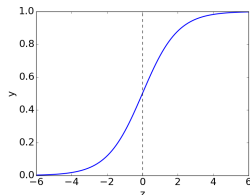


- The loss function hates when you make correct predictions with high confidence!
- If $t = 1$, it's more unhappy about $y = 10$ than $y = 0$.

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 **sigmoidal**, or S-shaped, function:

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



- A linear model with a logistic nonlinearity is known as **log-linear**:

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

$$y = \sigma(z)$$

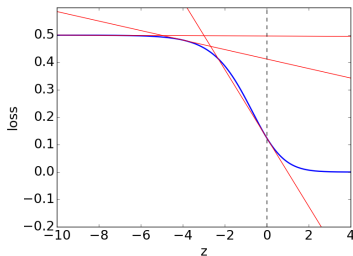
$$\mathcal{L}_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2.$$

- Used in this way, σ is called an **activation function**, and z is called the **logit**.

Attempt 3: Logistic Activation Function

The problem:

(plot of \mathcal{L}_{SE} as a function of z)

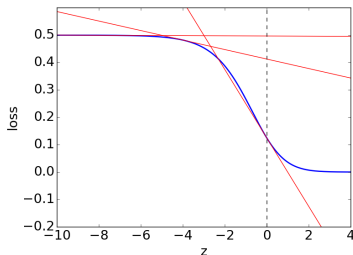


$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w_j}$$
$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{L}}{\partial w_j}$$

Attempt 3: Logistic Activation Function

The problem:

(plot of \mathcal{L}_{SE} as a function of z)



$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial w_j}$$
$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{L}}{\partial w_j}$$

- In gradient descent, a small gradient (in magnitude) implies a small step.
- If the prediction is really wrong, shouldn't you take a large step?

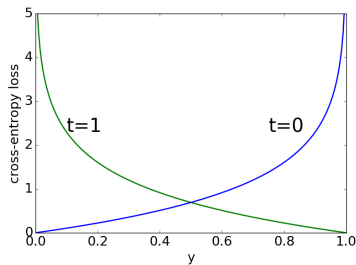
Logistic Regression

- 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.

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** captures this intuition:

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



Logistic Regression

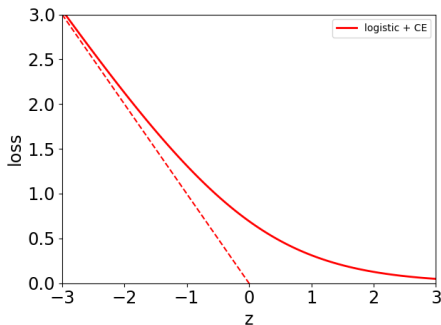
Logistic Regression:

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

$$y = \sigma(z)$$

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

$$\mathcal{L}_{\text{CE}} = -t \log y - (1 - t) \log(1 - y)$$



[[gradient derivation in the notes]]

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.

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

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.

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

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

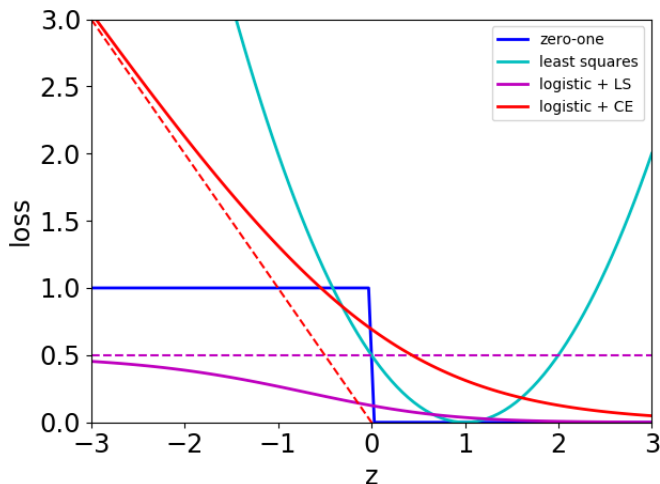
$$\mathcal{L}_{\text{LCE}}(z, t) = \mathcal{L}_{\text{CE}}(\sigma(z), t) = t \log(1 + e^{-z}) + (1 - t) \log(1 + e^z)$$

- Numerically stable computation:

$$E = t * \text{np.logaddexp}(0, -z) + (1-t) * \text{np.logaddexp}(0, z)$$

Logistic Regression

Comparison of loss functions:



Logistic Regression

Comparison of gradient descent updates:

- Linear regression:

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

- Logistic regression:

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

Logistic Regression

Comparison of gradient descent updates:

- Linear regression:

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

- Logistic regression:

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

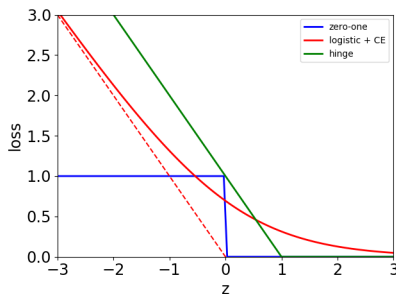
- Not a coincidence! These are both examples of **matching loss functions**, but that's beyond the scope of this course.

Hinge Loss

- Another loss function you might encounter is **hinge loss**. Here, we take $t \in \{-1, 1\}$ rather than $\{0, 1\}$.

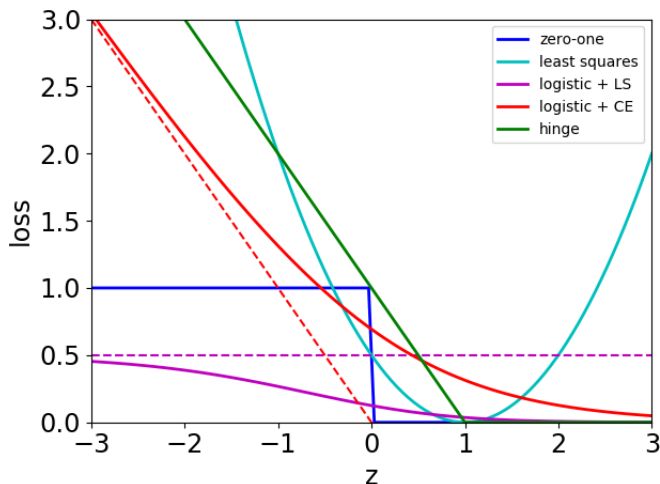
$$\mathcal{L}_H(y, t) = \max(0, 1 - ty)$$

- This is an **upper bound** on 0-1 loss (a useful property for a surrogate loss function).
- A linear model with hinge loss is called a **support vector machine**. You already know enough to derive the gradient descent update rules!
- Very different motivations from logistic regression, but similar behavior in practice.



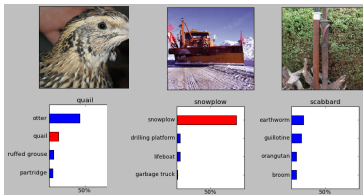
Logistic Regression

Comparison of loss functions:



Multiclass Classification

- What about classification tasks with more than two categories?



Multiclass Classification

- Targets form a discrete set $\{1, \dots, K\}$.
- It's often more convenient to represent them as **one-hot vectors**, or a **one-of-K encoding**:

$$\mathbf{t} = \underbrace{(0, \dots, 0, 1, 0, \dots, 0)}_{\text{entry } k \text{ is } 1}$$

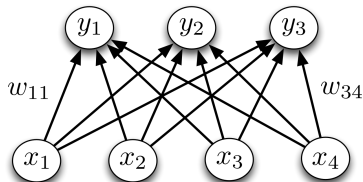
Multiclass Classification

- Now there are D input dimensions and K output dimensions, so we need $K \times D$ weights, which we arrange as a **weight matrix \mathbf{W}** .
- Also, we have a K -dimensional vector **\mathbf{b}** of biases.
- Linear predictions:

$$z_k = \sum_j w_{kj} x_j + b_k$$

- Vectorized:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$



Multiclass Classification

- A natural activation function to use is the **softmax function**, a multivariable generalization of the logistic function:

$$y_k = \text{softmax}(z_1, \dots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

- The inputs z_k are called the **logits**.
- Properties:
 - Outputs are positive and sum to 1 (so they can be interpreted as probabilities)
 - If one of the z_k 's is much larger than the others, $\text{softmax}(\mathbf{z})$ is approximately the argmax . (So really it's more like "soft- argmax ".)
 - **Exercise:** how does the case of $K = 2$ relate to the logistic function?
- Note: sometimes $\sigma(\mathbf{z})$ is used to denote the softmax function; in this class, it will denote the logistic function applied elementwise.

Multiclass Classification

- If a model outputs a vector of class probabilities, we can use cross-entropy as the loss function:

$$\begin{aligned}\mathcal{L}_{\text{CE}}(\mathbf{y}, \mathbf{t}) &= - \sum_{k=1}^K t_k \log y_k \\ &= -\mathbf{t}^\top (\log \mathbf{y}),\end{aligned}$$

where the log is applied elementwise.

- Just like with logistic regression, we typically combine the softmax and cross-entropy into a **softmax-cross-entropy** function.

Multiclass Classification

- Multiclass logistic regression:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathbf{y} = \text{softmax}(\mathbf{z})$$

$$\mathcal{L}_{\text{CE}} = -\mathbf{t}^\top (\log \mathbf{y})$$

- **Tutorial:** deriving the gradient descent updates

$$\frac{\partial \mathcal{L}_{\text{CE}}}{\partial \mathbf{z}} = \mathbf{y} - \mathbf{t}$$

Convex Functions

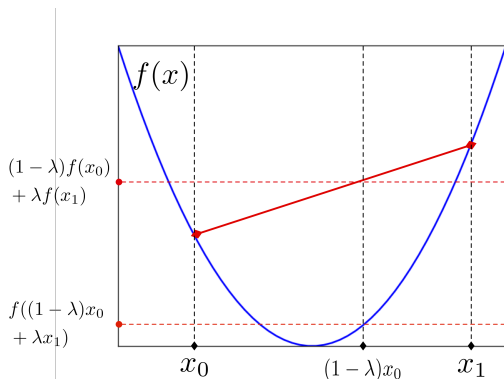
- Recall: a set \mathcal{S} is convex if for any $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$,

$$(1 - \lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$

- A function f is **convex** if for any $\mathbf{x}_0, \mathbf{x}_1$ in the domain of f ,

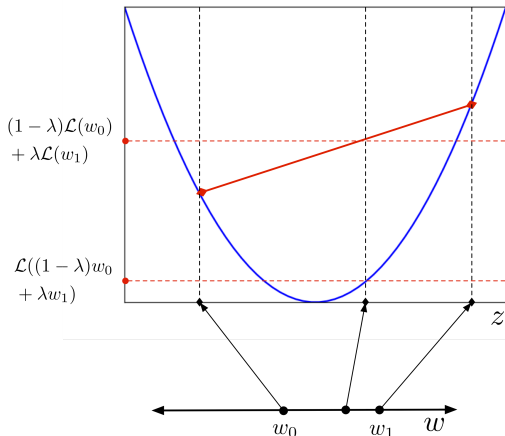
$$f((1 - \lambda)\mathbf{x}_0 + \lambda\mathbf{x}_1) \leq (1 - \lambda)f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1)$$

- Equivalently, the set of points lying above the graph of f is convex.
- Intuitively: the function is bowl-shaped.



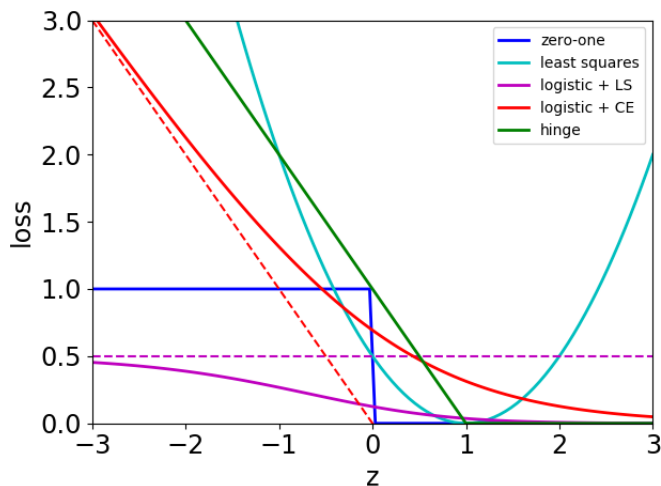
Convex Functions

- We just saw that the least-squares loss function $\frac{1}{2}(y - t)^2$ is convex as a function of y
- For a linear model, $z = \mathbf{w}^\top \mathbf{x} + b$ is a linear function of \mathbf{w} and b . If the loss function is convex as a function of z , then it is convex as a function of \mathbf{w} and b .



Convex Functions

Which loss functions are convex?



Convex Functions

Why we care about convexity

- All critical points are minima
- Gradient descent finds the optimal solution (more on this in a later lecture)

Gradient Checking

- We've derived a lot of gradients so far. How do we know if they're correct?
- Recall the definition of the partial derivative:

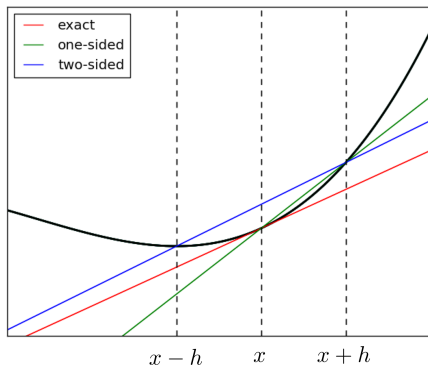
$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i, \dots, x_N)}{h}$$

- Check your derivatives numerically by plugging in a small value of h , e.g. 10^{-10} . This is known as **finite differences**.

Gradient Checking

- Even better: the two-sided definition

$$\frac{\partial}{\partial x_i} f(x_1, \dots, x_N) = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_N) - f(x_1, \dots, x_i - h, \dots, x_N)}{2h}$$



Gradient Checking

- Run gradient checks on small, randomly chosen inputs
- Use double precision floats (not the default for most deep learning frameworks!)
- Compute the **relative error**:

$$\frac{|a - b|}{|a| + |b|}$$

- The relative error should be very small, e.g. 10^{-6}

Gradient Checking

- Gradient checking is really important!
- Learning algorithms often appear to work even if the math is wrong.
- **But:**
 - They might work much better if the derivatives are correct.
 - Wrong derivatives might lead you on a wild goose chase.
- If you implement derivatives by hand, gradient checking is the single most important thing you need to do to get your algorithm to work well.