Supervised classification

- k-nearest neighbors
- Decision boundaries
- Classifiers based on linear models
- Softmax classifier
- Maximum-likelihood
- Support vector machines
- Hinge loss function

Introduction

- In this lecture, we focus on techniques to do supervised *classification*.
- We'll focus on the application of image classification, where convolutional neural networks are very appropriate.
- Regression problems (i.e., predicting a continuous output) as well as unsupervised and reinforcement learning problems are of great interest, but for the purposes of training convolutional neural networks, we'll start with the supervised classification problem.

Supervised classification

- In supervised classification, our goal is to take an input data point, $\mathbf{x} \in \mathbb{R}^n$, and assign it to one of k classes.
- Typically, this is achieved by calculating a *score* if the input was in each class. The score may be a probability, like we saw in the maximum likelihood example of the machine learning basics lecture.
- The class with the highest score is then the chosen (or assigned) class.
- There are several approaches that can be used to train supervised classifiers. In this class, what we'll do is focus on arriving at a reasonable *loss* function. After this, we'll introduce a general way to optimize the parameters with respect to these functions using gradient descent.
- With a focus on the formulation of cost functions and problem set up, with the goal of using gradient descent later on, we won't dive as deeply into some problems (e.g., we won't formulate the support vector machine as a convex optimization problem).

Supervised classification

- It is also worth noting that the softmax classifiers and SVM can be used at the *output* of a feedforward or convolutional neural network. (i.e., they'll be useful for later on.)
- In recurrent neural networks, typically a linear readout is used.

Back to supervised classification

Consider a setup that is the same as the maximum-likelihood classifier of the previous lecture. Imagine you were given a training set of input vectors, $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\}$ and their corresponding classes, $\{y^{(1)}, y^{(2)}, \ldots, y^{(m)}\}$.

Now imagine that you were also given a new data point, $\mathbf{x}^{\text{new}}$. We found a way to classify through a probabilistic model, where we had to learn parameters, but isn't there a simpler way to classify that doesn't involve very much "machine learning" machinery?

k-nearest neighbors

Intuitively, k-nearest neighbors says to find the k closest points (or nearest neighbors) in the training set, according to an appropriate metric. Each of its k nearest neighbors then vote according to what class it is in, and $\mathbf{x}^{\text{new}}$ is assigned to be the class with the most votes.

- A common metric is the Euclidean (or ℓ^2) norm. Under this norm, the distance to training dataset ponit $\mathbf{x}^{(i)} \in \mathbb{R}^N$ is given by:

$$\begin{aligned} d_i &= \sqrt{\sum_{j=1}^{N} (x_j^{\text{new}} - x_j^{(i)})^2} \\ &= \left\| \mathbf{x}^{\text{new}} - \mathbf{x}^{(i)} \right\| \end{aligned}$$

- It is possible to use other norms.

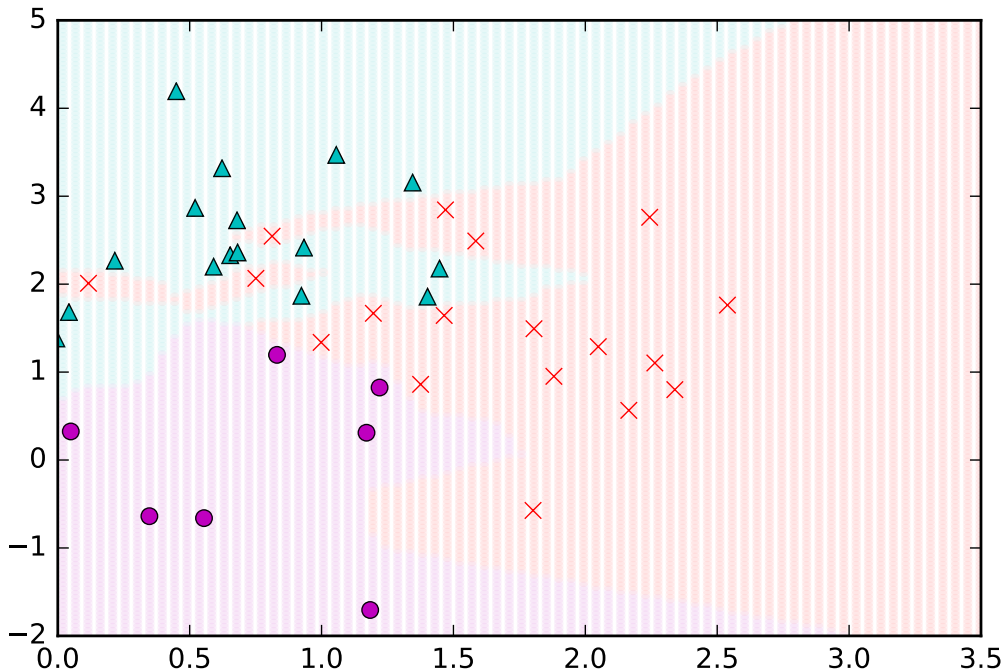
k-nearest neighbors, more formally

- Choose an appropriate distance metric, $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$, returning the distance between $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$. E.g., $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left\| \mathbf{x}^{(i)} - \mathbf{x}^{(j)} \right\|$.
- Choose the number of nearest neighbors, k.
- Take desired test data point, $\mathbf{x}^{\text{new}}$, and calculate $d(\mathbf{x}^{\text{new}}, \mathbf{x}^{(i)})$ for $i = 1, \ldots, m$.
- With $\{c_1, \ldots, c_k\}$ denoting the k indices corresponding to the k smallest $d(\mathbf{x}^{\text{new}}, \mathbf{x}^{(i)})$, classify $\mathbf{x}^{\text{new}}$ as the class that occurs most frequently amongst $\{y^{c_1}, \ldots, y^{c_k}\}$. If there is a tie, any of the tying classes may be selected.

Visualizing k-nearest neighbors

k-nearest neighbors boils down to the following image:

- The decision boundaries are nonlinear.
- The algorithm is nonparametric.
- Practically speaking, how do we select k and $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$? These are hyperparameters, and ought to be optimized by using k-fold cross validation.



k-nearest neighbors

k-nearest neighbors has several pros and cons. First, the pros:

- No training time.

Next, the cons:

- Requires caching the entire training set, which could be impractical if large.
- Is computationally expensive on testing new data.
- The curse of dimensionality may be at play.
- The data representation is very important.

This list of cons is considerable. However, we can be address the dataset caching and computational expense by learning *decision boundaries*. These boundaries separate the input space into separate regions (like k-nearest neighbors), but parametrize these boundaries. If there are c classes, it will be possible to classify this data point through c calculations, as opposed to m.

Decision boundaries

We let $a_i(\mathbf{x})$ denote the "score" that $\mathbf{x}$ is in class i. If there are c classes, we have c functions $a_1(\cdot), \ldots, a_c(\cdot)$.

- A point $\mathbf{x}$ is assigned to class i if $a_i(\mathbf{x}) > a_j(\mathbf{x})$ for all $i \neq j$.
- The *decision boundary* between class i and j are all points for which $a_i(\mathbf{x}) = a_j(\mathbf{x})$.

Decision boundaries

Example 1: Let $\mathbf{x} \in \mathbb{R}^N$. Consider the score $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + w_i$. This type of classifier results in *linear* decision boundaries, because $a_i(\mathbf{x}) = a_j(\mathbf{x})$ implies:

$$(\mathbf{w}_i - \mathbf{w}_j)^T \mathbf{x} + (w_i - w_j) = 0$$

This is the equation of an $N-1$ dimensional hyperplane in $\mathbb{R}^N$.

Example 2: Consider a matrix, $\mathbf{W}$, defined as:

$$\begin{bmatrix} - \mathbf{w}_1^T - \\ \vdots \\ - \mathbf{w}_c^T - \end{bmatrix}$$

Then, $\mathbf{W} \in \mathbb{R}^{c \times N}$. Let $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{b}$ is a vector of bias terms. Then $\mathbf{y} \in \mathbb{R}^c$ is a vector of scores, with its ith element corresponding to the score of $\mathbf{x}$ being in class i. The chosen class corresponds to the index of the highest score in $\mathbf{y}$.

Softmax function

There are several instances when the scores should be normalized. This occurs, for example, in instances where the scores should be interpreted as probabilities. In this scenario, it is appropriate to apply the *softmax* function to the scores.

The softmax function transforms the class score, $\mathrm{softmax}_i(\mathbf{x})$, so that:

$$\mathrm{softmax}_i(\mathbf{x}) = \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}}$$

for $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + w_i$ and c being the number of classes.

If we let $\theta = \{\mathbf{w}_j, w_j\}_{j=1,\ldots,c}$, then $\mathrm{softmax}_i(\mathbf{x})$ can be interpreted as the probability that $\mathbf{x}$ belongs to class i. That is,

$$\Pr(y^{(j)} = i | \mathbf{x}^{(j)}, \theta) = \mathrm{softmax}_i(\mathbf{x}^{(j)})$$

Softmax classifier

Although we know the softmax function, how do we specify the *objective* to be optimized with respect to θ?

One intuitive heuristic is that we should choose the parameters, θ, so as to maximize the likelihood of having seen the data. Assuming the samples, $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$ are iid, this corresponds to maximizing:

$$\begin{aligned} p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}, y^{(1)}, \ldots, y^{(m)} | \theta) &= \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, y^{(i)} | \theta) \\ &= \prod_{i=1}^{m} p(\mathbf{x}^{(i)} | \theta) p(y^{(i)} | \mathbf{x}^{(i)}, \theta) \end{aligned}$$

Softmax classifier

We next note that:

$$\begin{aligned}
\arg\max_\theta \prod_{i=1}^m p(\mathbf{x}^{(i)}|\theta)p(y^{(i)}|\mathbf{x}^{(i)},\theta) &= \arg\max_\theta \prod_{i=1}^m p(y^{(i)}|\mathbf{x}^{(i)},\theta) \\
&= \arg\max_\theta \sum_{i=1}^m \log p(y^{(i)}|\mathbf{x}^{(i)},\theta) \\
&= \arg\max_\theta \sum_{i=1}^m \log \text{softmax}_{y^{(i)}}(\mathbf{x}^{(i)}) \\
&= \arg\max_\theta \sum_{i=1}^m \left(a_{y^{(i)}}(\mathbf{x}^{(i)}) - \log \sum_{j=1}^c e^{a_j(\mathbf{x})} \right) \\
&= \arg\min_\theta \frac{1}{m} \sum_{i=1}^m \left(\log \sum_{j=1}^c e^{a_j(\mathbf{x})} - a_{y^{(i)}}(\mathbf{x}^{(i)}) \right)
\end{aligned}$$

Solving this minimization results in the the *softmax classifier*. (Note, we haven't talked yet about how to choose θ to solve this minimization problem. We've only formulated the objective.) Note that we'll typically normalize the loss by the number of training examples, $\frac{1}{m}$.

Overflow of softmax

A potential problem when implementing a softmax classifier is overflow.

- If $a_i(\mathbf{x}) \gg 0$, then $e^{a_i(\mathbf{x})}$ may be very large, and numerically overflow and / or result to numerical imprecision.
- Thus, it is standard practice to normalize the softmax function as follows:

$$\begin{aligned}
\text{softmax}_i(\mathbf{x}) &= \frac{e^{a_i(\mathbf{x})}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\
&= \frac{ke^{a_i(\mathbf{x})}}{k\sum_{j=1}^{c} e^{a_j(\mathbf{x})}} \\
&= \frac{e^{a_i(\mathbf{x})+\log k}}{\sum_{j=1}^{c} e^{a_j(\mathbf{x})+\log k}}
\end{aligned}$$

- A sensible choice of k is so that $\log k = -\max_i a_i(\mathbf{x})$, making the maximal argument of the exponent 0.

Softmax classifier: intuition

When optimizing likelihoods, we typically work with the "log likelihood." When applying the softmax, we interpret its output as the probability of a class.

$$\begin{aligned} \log \Pr(y = i|\mathbf{x}) &= \log \text{softmax}_i(\mathbf{x}) \\ &= a_i(\mathbf{x}) - \log \sum_{j=1}^{c} \exp(a_j(\mathbf{x})) \end{aligned}$$

When maximizing this, the term $a_i(\mathbf{x})$ is made larger, and the term $\log \sum_j \exp(a_j(\mathbf{x}))$ is made smaller. The latter term can be approximated by $\max_j a_j(\mathbf{x})$. (Why?)

We consider two scenarios:

- If $a_i(\mathbf{x})$ produces the largest score, then the log likelihood is approximately 0.
- If $a_j(\mathbf{x})$ produces the largest score for $j \neq i$, then $a_i(\mathbf{x}) - a_j(\mathbf{x})$ is negative, and thus the log likelihood is negative.

Support vector machine: introduction

Another common decision boundary classifier is the support vector machine (SVM).

Informally, the SVM finds a boundary that maximizes the *margin*, or intuitively the "gap" between the boundary and the data points. The fundamental idea here is that if a point is further away from the decision boundary, there ought to be greater *confidence* in classifying that point.

Support vector machine: introduction

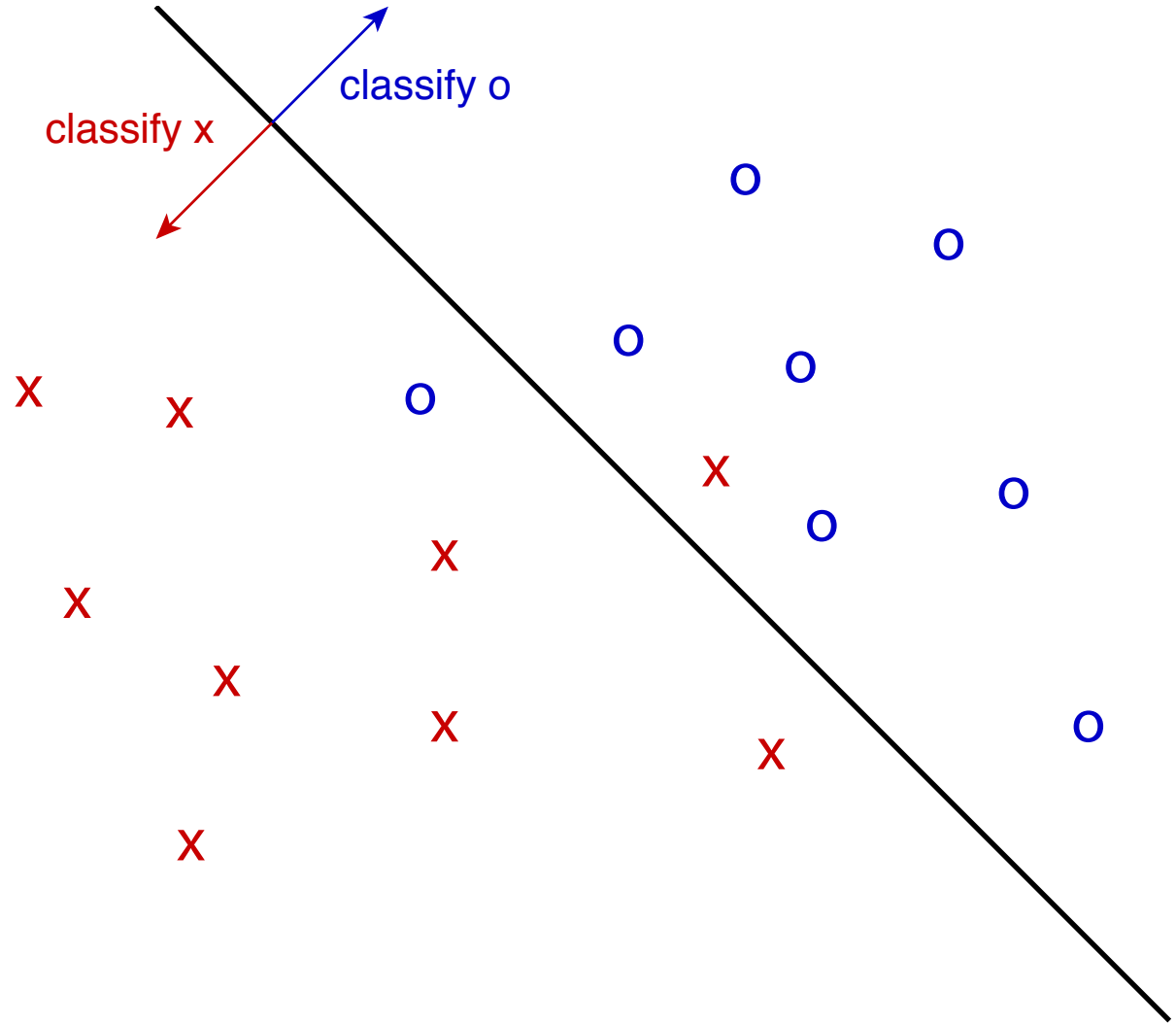
The SVM is a commonly used and has much theory behind it. A typical machine learning class will formulate the SVM as a convex optimization problem. However, this is beyond the scope of this class.

Instead, we'll talk about the SVM at a very high-level using a soft-margin "hinge loss" and provide appropriate intuitions. We'll focus on linear SVMs and will *not* touch on kernels. Please look into a machine learning class for more information about the SVM.

(Like the softmax classifier, the SVM may be used at the output of a neural network.)

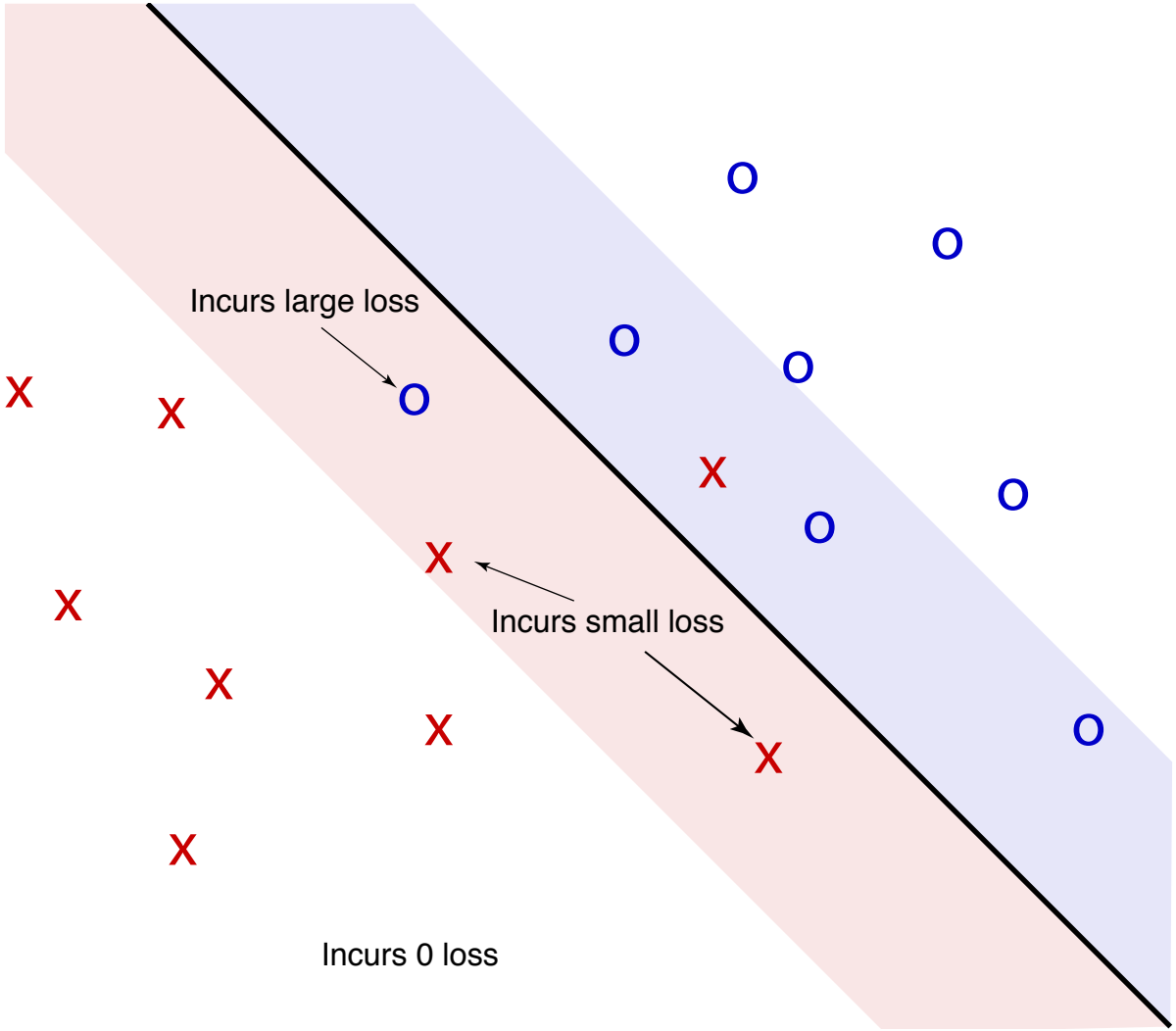
Support vector machine: intuition

The following picture is appropriate to have in mind for the SVM:



Support vector machine: intuition

The following picture is appropriate to have in mind for the SVM:



The hinge loss function

The hinge loss is standardly defined for a binary output $y^{(i)} \in \{-1, 1\}$. If $y^{(i)} = 1$, then we would like $\mathbf{w}^T\mathbf{x}^{(i)} + b$ to be large and positive. If $y^{(i)} = -1$, then we would like $\mathbf{w}^T\mathbf{x}^{(i)} + b$ to be large and negative. The larger $a_i(\mathbf{x}^{(j)})$ is in the right direction, the larger the margin.

In this scenario, the hinge loss is for $x^{(i)}$ being in class $y^{(i)}$ is:

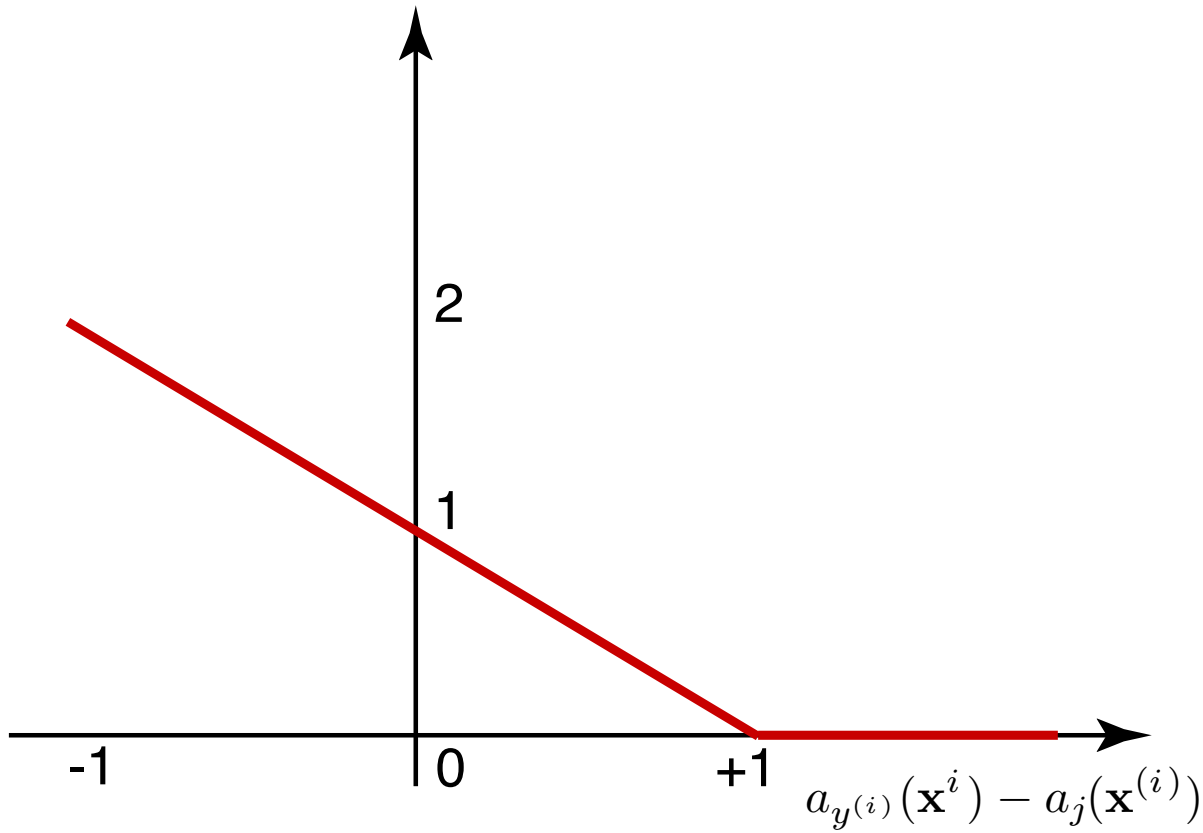
$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \max(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b))$$

This is a loss, and hence something we wish to minimize. There are a few things to notice about the form of this function.

- If $\mathbf{w}^T\mathbf{x}^{(i)} + b$ and $y^{(i)}$ have the same sign, indicating a correct classification, then $0 \leq \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) \leq 1$.
  - The error will be zero if $\mathbf{w}^T\mathbf{x}^{(i)} + b$ is large, corresponding to a large margin.
  - The error will be nonzero if $\mathbf{w}^T\mathbf{x}^{(i)} + b$ is small, corresponding to a small margin.
- If $\mathbf{w}^T\mathbf{x}^{(i)} + b$ and $y^{(i)}$ have opposite signs, then the hinge loss is non-negative, i.e., $\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = 1 + |\mathbf{w}^T\mathbf{x}^{(i)} + b|$.

Hinge loss intuition

The intuition of the prior slide is that the hinge loss is greatest for misclassifications, and the greater the error in misclassification, the worse the loss. For correct classifications, the loss will be zero only if there is a large enough margin.



Hinge loss extension

An extension of the hinge loss to multiple potential outputs is the following loss:

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

for $a_j(\mathbf{x}^{(i)}) = \mathbf{w}_j^T \mathbf{x}^{(i)}$. Some intuitions, for the scenario that there are c classes:

- When the correct class achieves the highest score, $a_{y^{(i)}}(\mathbf{x}^{(i)}) \geq a_j(\mathbf{x}^{(i)})$ for all $j \neq y^{(j)}$, then $a_j(\mathbf{x}^{(i)}) - a_{y_{(i)}}(\mathbf{x}^{(i)}) \leq 0$ and

$$0 \leq \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) \leq c - 1$$

- When an incorrect class, class i, achieves the highest score, then $a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}) \geq 0$ and has the potential to be large.
- In both scenarios, it is still desirable to make the correct margins larger and the incorrect margins smaller.

The SVM cost function

If we let $\theta = \{\mathbf{w}_j\}_{j=1,\ldots,c}$, where there are c classes, we can now formulate the SVM optimization function. In particular, we want to minimize the hinge loss across all training examples. Then, to optimize θ for a linear kernel and hinge loss, we solve the following minimization problem:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})$$

which, for the sake of completeness, can be writen as:

$$\arg\min_{\theta} \frac{1}{m} \sum_{i=1}^{m} \sum_{j \neq y^{(i)}} \max(0, 1 + a_j(\mathbf{x}^{(i)}) - a_{y^{(i)}}(\mathbf{x}^{(i)}))$$

Optimization

Equipped with these objective functions, we now turn to solving the optimization problems. Note that in the machine learning basics lecture, we could set derivatives equal to zero because our costs were quadratic and had closed-form solutions. The introduction of nonlinearities means that these problems no longer have closed-form solutions.

This is apparent through a simple example: consider a solution where the SVM loss is equal to zero. Another solution is to multiply all the weights by two, and the loss would still be zero.

However, these problems are convex. While there are several different ways to carry out optimization given a unique scenario, we will perform optimization by using gradient (or subgradient) descent.

(Even if the problem isn't convex, gradient descent can be very useful. In fact, this is how we will train nonconvex neural networks.)