SVMs and Neural Networks

CS 360 Machine Learning

Week 9, Day 1

March 26, 2024

Contents

1 Support Vector Machines (SVMs)

Recall that the SVM optimization problem is:

$$\min_{\vec{w},b} \ \frac{1}{2}||\vec{w}||^2$$

$$\text{s.t. } \ y_i(\vec{w}\cdot\vec{x}_i + b) - 1 \geq 0 \text{ for } i = 1, ..., n$$

Recall also that we require that our training data examples are classified perfectly by this and can't be on the boundary. In order to solve this, we write a Lagrangian where $\vec{\alpha}$ is a vector of Lagrange multipliers, one for every training example:

$$\mathcal{L}(\vec{w}, b, \vec{\alpha}) = \frac{1}{2}||\vec{w}||^2 - \sum_{i=1}^{n}\alpha_i\Big[y_i(\vec{w}\cdot\vec{x}_i + b) - 1\Big]$$

In order to solve this, we take the gradient with respect to all three of these variables, $\vec{w}, b, \vec{\alpha}$, set to 0, and solve. Taking the derivative with respect to α makes sure that the gradients with respect to both of these functions point in the same direction.

First we take the gradient with respect to $\vec{w}$:

$$\nabla_{\vec{w}}\mathcal{L}(\vec{w}, b, \vec{\alpha}) = \vec{w} - \sum_{i=1}^{n}\alpha_i y_i \vec{x}_i = \vec{0}$$

Setting this equal to 0 and solving, we get:

$$\vec{w} = \sum_{i=1}^{n}\alpha_i y_i \vec{x}_i$$

which reminds us of the weight updates for the perceptron algorithm, but we now have an α_i with a dependency on the specific example i.

When our constraints on the original optimization problem are active we are considering a support vector example, i.e. $\alpha_i > 0$ if $\vec{x}_i$ is a support vector. $\alpha_i = 0$ for non-support vector examples.

If we take the partial derivative of the Lagranian with respect to b, we get:

$$\frac{\partial \mathcal{L}(\vec{w}, b, \vec{\alpha})}{\partial b} = \sum_{i=1}^{n} \alpha_i y_i = 0$$

If we consider only the per-example Lagrange multipliers for the positively classified examples versus the negatively classified ones, we find that the sum over these categories are equal:

$$\sum_{i:y_i=-1} \alpha_i = \sum_{i:y_i=1} \alpha_i$$

Suppose that we have two positive and one negative examples as support vectors. One way to find the hyperplane is to connect the two same classified examples and connect them with a line. Taking a perpendicular from the other example to that line, and look at the perpendicular bisector to this line, that's the hyperplane.

Using the gradient with respect to $\vec{w}$ we can plug the above back into the Lagrangian and solve the dual optimization problem:

$$\max_{\vec{\alpha}} W(\vec{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \vec{x}_i \cdot \vec{x}_j$$

$$\text{s.t. } \alpha_i \geq 0$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

This distills the problem down to the hard problem of optimizing the dot product between pairs of vectors ($\vec{x}_i \cdot \vec{x}_j$) which could be replaced by any other feature combination. This allows a lot of flexibility in the (dual) optimization version of SVM.

1.1 Kernel idea

By solving the dual form of the problem, the key computational step becomes the dot product (as seen above). This gives us a notion of similarity between examples in a linear version of SVMs. This is sort of the opposite of a distance function – the distance is small if two points are similar, but in this case the similarity is large if two features are similar. Replacing this notion of similarity with other notions is known as the *kernel trick*, and allows us to capture other non-linear separating options. We can also think of this as (and this is sometimes implemented by) first transforming the features into a new space and then taking a separating hyperplane in that new space; all of this is captured by the *kernel function* which replaces the dot product in the optimization function.

One commonly used kernel is the radial basis function (RBF) kernel:

$$K(\vec{x}, \vec{z}) = \exp\left(-\frac{||\vec{x} - \vec{z}||^2}{2\sigma^2}\right)$$

This is often re-parameterized by $\gamma = \frac{1}{2\sigma^2}$ giving us:

$$K(\vec{x}, \vec{z}) = \exp\left(-\gamma ||\vec{x} - \vec{z}||^2\right)$$

By using this kernel, things that are very far away from the separating hyperplane are weighted similarly to points that are far, but not quite as far away. The exponential decay makes sure that very far away points aren't overweighted in the optimization. We can think about this as a neighborhood of influence.

1.2 Soft-margin SVMs

Soft-margin SVMs handle the case where there might be outliers that don't exactly share the classification of the other training data. Adding a "flexible margin" parameter to the objective function allows some of the points to be misclassified. This gives a slightly modified optimization problem:

$$\min_{\xi,\vec{w},b} \frac{1}{2}||\vec{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

$$\text{s.t. } y_i(\vec{w}\cdot\vec{x}_i + b) \geq 1 - \xi_i \text{ for } i = 1, ..., n$$

$$\xi_i \geq 0 \text{ for } i = 1, ..., n$$

1.3 Meta-optimization process

The idea of meta-optimization is an incremental SVM creation process. Using the idea that new examples can only bring the margin tighter, this process works by choosing a subset of examples and running the optimization to determine α_i values. The α_i values that are 0 are identified – the $\vec{x}_i$ points associated with these α_i values cannot be support vectors in the final solution, so they can be discarded. Once those points are discarded, new example points are added and the process is repeated.

1.4 Disadvantages of SVMs

SVMs are powerful general purpose models, and for a long time were the "go-to" model of choice. However, there are a lot of unknowns which can make it hard to optimize the model. It's hard to know in advance what a good kernel function choice is, and when there are a lot of correlated features that can be hard to handle. The final model is also not interpretable.

While neural networks have replaced SVMs in many cases as the standard model choice, they do not solve all of these problems!

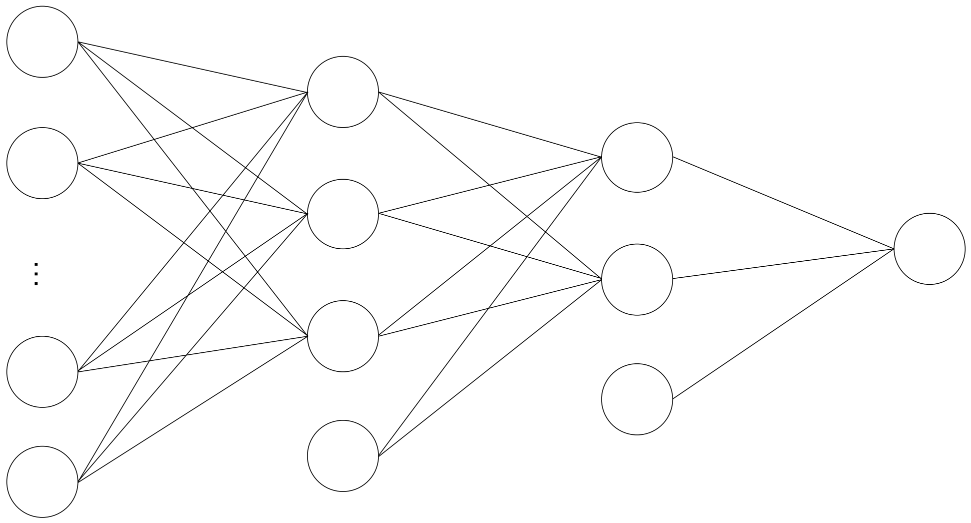
2 Neural networks

Neural networks are heavily used in modern machine learning. It's possible to write them without really understanding them in a few lines of code – but it's still important to understand the methods we're using! This will be the main topic for the rest of the course.

Neural networks are inspired by biological neurons, as the perceptron model was, but now instead of modeling a single neuron, the idea is to model a whole brain. What people were trying to do with early neural networks was taking input like an image and make some determination about the photo, e.g., is it a person with glasses, smiling, etc. The first idea of this was to project this input data down into a lower dimension where it's easier for the model to understand. This could help to combine different features to, hopefully, understand these complex features like glasses. These lower dimensional projections are the *hidden layers* of a neural network, between the input and output layers. In Sara's research, she considers this for genetic data, where the input layer takes DNA information and the output gives information about population sizes.

The perceptron model can be interpreted as a simple neural network. As we discussed, the perceptron on its own can't learn the XOR function... but if multiple perceptrons are combined together, then it can! This solved a key concern of early AI research. However, another concern came in training multi-layer (deep) neural networks – how can the needed weights for these models be determined? This contributed to a second decline in AI (a second "AI winter"). But in the mid-2000s (2006) there was a breakthrough in training neural networks that allowed them to be trained more easily, and so now they are heavily used.

Neural networks are universal approximators – they can approximate any function! We will be interested in approximating a function that maps from our inputs to our outputs. We'll still have some true label and some predicted label and will try to minimize the loss of a function evaluating the mapping between the output and the true output. We'll use gradient decent like approaches to minimize this loss.



Above is a standard representation of a fully-connected neural network. We think of the data as moving through the network from left to right, with each node on the left representing one of p features of the input from x_1 to x_p with the final node representing the fake 1. This first column of nodes on the left is known as the *input layer*. The single node all the way on the right is known as the *output layer*. The columns of nodes from left to right between the input and output layers are known as the *hidden layers* of the neural network. We'll use superscripts to number the hidden layers, so information associated with the ith hidden layer will have the superscript (i), with counting generally starting at (1) such that the second from the left column of nodes above is known as the 1st hidden layer, and so on counting to the right. Within each layer (column of nodes), we'll count nodes starting at 1 from the top to the bottom.

In each hidden layer, the input to that layer is the output of the previous one, such that the ith node in the first hidden layer holds the value denoted $h_i^{(1)}$, which is determined based on the weights along each edge from the first layer to the second layer. The value of the ith node in the kth hidden layer is similarly denoted $h_i^{(k)}$. The weight from the first node in the first layer to the first node in the second layer (first hidden layer) is denoted $w_{11}^{(1)}$, the weight from the first node in the first layer to the second node in the first hidden layer is denoted $w_{12}^{(1)}$, and so on such that the weight from the ith node in the input layer to the jth node in the first hidden layer is denoted $w_{ij}^{(1)}$. The vector of weights incoming from the input layer to the first node in the hidden layer (not including the bias term) will be denoted $\vec{w}_1^{(1)}$, i.e.:

$$\vec{w}_1^{(1)} = \begin{bmatrix} w_{11}^{(1)} \\ w_{21}^{(1)} \\ w_{31}^{(1)} \\ \vdots \\ w_{p1}^{(1)} \end{bmatrix}$$

By convention, the final (lowest) node in the layer (column) will represent the fake 1, and the edge from that node will represent the bias term. The bias edge from the fake 1 in the first layer to the first node in the second layer is denoted $b_1^{(1)}$. Since by convention, the bias will always be the last (bottom) node, the subscript on b will denote the node number in the first hidden layer it connects to and omit the indication of the position of the fake 1 in the first layer.

Using this notation, we can now write the value of the first node in the first hidden layer as:

$$h_1^{(1)}(\vec{x}) = a\left(\vec{w}_1^{(1)} \cdot \vec{x} + b_1^{(1)}\right)$$

where a is a linear activation function. If we only had one hidden layer with one node, we can think about it as simply linear regression (without an activation function) or logistic regression if a is the logistic activation function. We can now think of linear and logistic regression as a small neural network!

We'll denote the first hidden layer as:

$$H^{(1)} = a\left(XW^{(1)} + b^{(1)}\right)$$

where X is an $n \times p$ matrix, W is of size $p \times p_1$, and b is a vector of length p_1 where p_1 is the number of nodes in the second layer (first hidden layer) not including the fake 1. In the above example network $p_1 = 3$ and $p_2 = 2$. We sometimes "broadcast" b to make it $n \times p_1$ by copying the bias entries. We don't know either W or b and we'll choose both of these using something called backpropogation, which is a form of gradient descent – we'll discuss this further in a later class.

Considering the second hidden layer, we now have:

$$H^{(2)} = a\left(H^{(1)}W^{(2)} + b^{(2)}\right)$$

Note that this uses the results of the first hidden layer to compute the results of the second. $H^{(1)}$ is $n \times p_1$, $W^{(2)}$ is $p_1 \times p_2$, and $b^{(2)}$ is $p_2 \times 1$.

This continues on until the final output layer in the small network above is:

$$\hat{\vec{y}} = a\left(H^{(2)}W^{(3)} + b^{(3)}\right)$$

The dimension of $\hat{\vec{y}}$ is $n \times 1$, i.e., it gives a prediction for each example. This is really just a series of linear regressions with activation functions applied!

Suppose that I had k classes. We haven't said exactly what $\hat{\vec{y}}$ is, but we usually think about it as a vector of probabilities. If there are multiple classes, then the resulting $\hat{\vec{y}}$ is $n \times k$ where it gives a probability per class for each of the n examples. One loss function to evaluate the neural network would then be cross entropy:

$$H(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log_k \hat{y}_k$$

All of our existing techniques for evaluating models will still apply to neural networks.

2.1 Activation functions

There are a number of options we can use for activation functions (a above). Three commonly used functions are sigmoid, tanh, and ReLU.

Definition 1 (sigmoid function)**.**

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

Definition 2 (hyperbolic tangent (tanh))**.**

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Definition 3 (rectified linear unit (ReLu))**.**

$$f(x) = \max(0, x)$$

ReLU has become the "go-to" standard choice for activation function. There are problems with both sigmoid and tanh; when you get very large positive or negative numbers then the gradient saturates – the derivative becomes 0 at the extremes and you're not getting useful input back. ReLU also has some problems where the signal can essentially 0 out. (See the slides for more positives and negatives of each choice.) Since ReLU is a standard go-to activiation function, we'll use that for the lab. Recall also our earlier discussion in this course about the use of mini-batches in the context of gradient descent – we'll also use mini-batches for the lab.