Applications:

- Binary classification
- Deep Neural Networks

Topics

- Stochastic Gradient Descent
- Linear Classification and the Perceptron Algorithm
- Multilayer perceptrons

Stochastic Gradient Descent

Last class, motivated by machine learning applications, we discussed how to apply gradient descent to solve the following optimization problem:

$$\text{minimize } \text{loss}((\underline{z}_i, \underline{y}_i); \underline{x}) = \text{minimize } \frac{1}{N}\sum_{i=1}^{N} \ell(\underline{m}(\underline{z}_i; \underline{x}) - \underline{y}_i) \qquad (L)$$

over the parameters $\underline{x}$ of a model $\underline{m}$ so that $\underline{m}(\underline{z}_i; \underline{x}) \approx \underline{y}_i$ for all training data input/output pairs $(\underline{z}_i, \underline{y}_i)$, $i = 1, \ldots, N$.

The bulk of our effort was spent on understanding how to compute the gradient of $\ell(\underline{m}(\underline{z}_i; \underline{x}) - \underline{y}_i)$ with respect to the model parameters $\underline{x}$, with a particular focus on models $\underline{m}$ that can be written as the following composition of models:

$$\begin{aligned}
\underline{o}_0 &= \underline{z}_i \\
\underline{o}_1 &= \underline{m}_1(\underline{o}_0; \underline{x}_1), \quad \underline{o}_1 \in \mathbb{R}^{p_1}, \ \underline{o}_0 \in \mathbb{R}^{p_0} \\
\underline{o}_2 &= \underline{m}_2(\underline{o}_1; \underline{x}_2), \quad \underline{o}_2 \in \mathbb{R}^{p_2}, \ \underline{o}_1 \in \mathbb{R}^{p_1} \qquad (DNN) \\
&\ \vdots \\
\underline{o}_L &= \underline{m}_L(\underline{o}_{L-1}; \underline{x}_L), \quad \underline{o}_L \in \mathbb{R}^{p_L}, \ \underline{o}_{L-1} \in \mathbb{R}^{p_{L-1}},
\end{aligned}$$

which is the structure of contemporary models used in machine learning called deep neural networks (we'll talk about these much more today). We made two key observations about the model structure that allowed us to effectively apply the matrix chain rule to compute gradients $\frac{\partial \ell}{\partial \underline{x}_1}, \ldots, \frac{\partial \ell}{\partial \underline{x}_L}$:

1) $\frac{\partial \ell}{\partial \underline{x}_j}$ only needs to compute partial derivatives of ℓ and layers $j, j+1, \ldots, L$;
2) If we start from layer L ($\frac{\partial \ell}{\partial \underline{x}_L}$) and work our way backwards we can
   (i) reuse previously computed partial derivatives, and
   (ii) save space on memory by exploiting that $\frac{\partial \ell}{\partial \underline{m}_L}$ is a row-vector.

The resulting algorithm is called backpropagation, and is a key enabling technology in modern machine learning: you will learn more about this in ESE 5460.

Now, despite all of this cleverness, when the model parameter vectors $\underline{x}_1, \ldots, \underline{x}_L$ and layer outputs $\underline{o}_1, \ldots, \underline{o}_L$ are very high dimensional (it is not uncommon for each $\underline{x}_i$ to have 100s of thousands or even millions of components), computing the gradient $D_{\underline{x}}\, \ell(\underline{m}(\underline{z}_i; \underline{x}) - \underline{y}_i)$ of a single term in the sum (L) can be quite costly. Add to that the fact that the number of data points N is often very large (order of millions in many settings), and we quickly run into some serious computational bottlenecks. And remember, this all just so we can run a single iteration of gradient descent. This may seem hopeless, but luckily, there is a very simple trick that lets us work around this problem: stochastic gradient descent.

Stochastic Gradient Descent (SGD) is the work horse algorithm of modern machine learning and has been rediscovered by various communities over the past 70 years, although it is usually credited to Robbins and Monro for a paper they wrote in 1951.

Key Idea: Since our loss function can be written as a sum over examples, i.e.

$$\text{(LL)} \qquad \text{loss}((\underline{z}_i, y_i); \underline{x}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\underline{m}(\underline{z}_i; \underline{x}) - y_i), \qquad \left(\text{loss}(\underline{x}) = \sum \ell_i(\underline{x})\right)$$

then the gradient is also a sum: $\nabla_{\underline{x}} \text{loss} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\underline{x}} \ell_i$. Therefore we expect each individual gradient $\nabla_{\underline{x}} \ell_i$ to have some useful information in it. SGD minimizes (LL) by following the gradient of a single randomly selected example (or a small batch of B randomly selected samples).

The SGD algorithm can be summarized as follows: Start with an initial guess $\underline{x}^{(0)}$, and at each iteration $k=0,1,2,\ldots$, do:

(i) Select an index $i \in \{1, \ldots, N\}$ at random
(ii) Update

$$\underline{x}^{(k+1)} = \underline{x}^{(k)} - s^{(k)} \nabla_{\underline{x}} \ell_i(\underline{x}^{(k)}) \qquad \text{(SGD)}$$

Using the gradient of only the i^{th} loss term $\ell_i(\underline{x}) = \ell(\underline{m}(\underline{z}_i; \underline{x}) - y_i)$.

As before, $s^{(k)} > 0$ is a step-size that can change as a function of the current iterate.

This method works shockingly well in practice and is computationally tractable as at each iteration, the gradient of only the i^{th} loss term needs to be computed. Modern versions of this algorithm replace step (i) with a mini-batch, i.e., by selecting B indices at random, and step (ii) replaces $\nabla_{\underline{x}} \ell_i(\underline{x}^{(k)})$ with the average gradient:

$$\frac{1}{B}\sum_{b=1}^{B} \nabla_{\underline{x}} \ell_b(\underline{x}). \qquad (\hat{G})$$

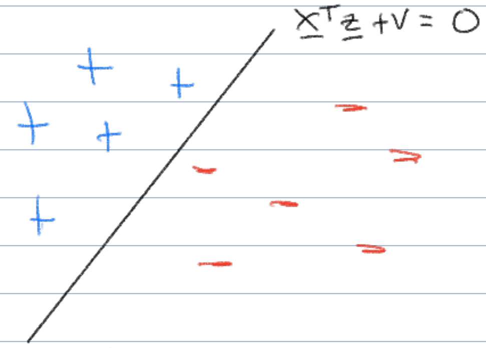
The overall idea behind why SGD works (take ESE 605U if you want to see a rigorous proof) is that while each individual update (SGD) may not be an accurate gradient for the overall loss function $\text{loss}(\underline{x})$, we are still following $\nabla_{\underline{x}} \text{loss}(\underline{x})$ "on average". This also explains why you may want to use a mini-batch B to compute a better gradient estimate $(\hat{G})$, as having more loss terms leads to a better approximation of the true gradient. The tradeoff though is that B becomes larger, computing $(\hat{G})$ is more computationally demanding.

Linear Classification and the Perceptron

An important problem in machine learning is that of binary classification. In one of the online case studies, you saw how to use least-squares to solve this problem. Here, we offer an alternative perspective that will lead us to one important historical reason for the emergence of deep neural networks.

The problem set up for linear binary classification is as follows. We are given a set of N vectors $\underline{z}_1, \dots, \underline{z}_N \in \mathbb{R}^n$ with associated binary labels $y_1, \dots, y_N \in \{-1, +1\}$. The objective in linear classification is to find an affine function $\underline{x}^T \underline{z} + v$, defined by unknown parameters $\underline{x} \in \mathbb{R}^n$ and $v \in \mathbb{R}$, that strictly separates the two classes. We can pose this as finding a feasible solution to the following linear inequalities:

$$\text{(LC)} \quad \begin{aligned} \underline{x}^T \underline{z}_i + v &> 0 \quad \text{if } y_i = +1 \\ \underline{x}^T \underline{z}_i + v &< 0 \quad \text{if } y_i = -1 \end{aligned}$$



The geometry of this problem is illustrated on the right. There are three key components:

1) The separating hyperplane $H = \{\underline{z} \in \mathbb{R}^n : \underline{x}^T \underline{z} + v = 0\}$. This is the set of vectors $\underline{z} \in \mathbb{R}^n$ that live on the subspace H, which is the solution set to the linear equation

$$\underline{x}^T \underline{z} = -v.$$

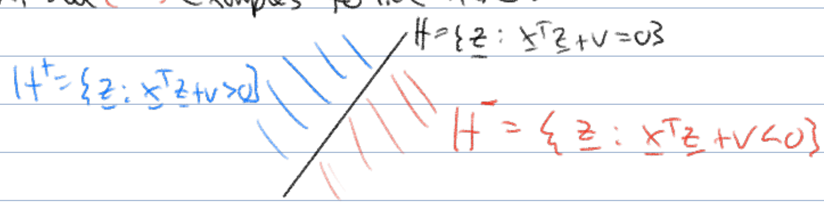
The coefficient matrix here is $\underline{x}^T \in \mathbb{R}^{1 \times n}$, and so rank $\text{Col}(\underline{x}^T) = 1$. This tells us that $\dim \text{Null}(\underline{x}^T) = \dim H = n-1$. In $\mathbb{R}^2$, this is the equation of a line:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} + v = x_1 z_1 + x_2 z_2 + v = 0 \iff z_2 = -\frac{x_1}{x_2} z_1 - \frac{v}{x_2}$$

In $\mathbb{R}^3$, this is the equation of a plane with normal vector $\underline{x}$ going through point v, and in $\mathbb{R}^n$ is called a hyper plane. A key feature of a hyperplane is that it splits $\mathbb{R}^n$ into two half-spaces, i.e., the subsets of $\mathbb{R}^n$ on either side.

2) The half-space $H^+ = \{\underline{z} \in \mathbb{R}^n : \underline{x}^T \underline{z} + v > 0\}$, which is the "half" of $\mathbb{R}^n$ for which $\underline{x}^T \underline{z} + v > 0$. We want all of our positive (+) examples to live here.

3) The half-space $H^- = \{\underline{z} \in \mathbb{R}^n : \underline{x}^T \underline{z} + v < 0\}$, which is the "half" of $\mathbb{R}^n$ for which $\underline{x}^T \underline{z} + v < 0$. We want all (−) examples to live here.



The problem of finding the parameters $(\underline{x}, v)$ defining the classifier in (LC) can be solved using linear programming, a kind of optimization algorithm that you'll learn about in ESE 3040 and ESE 6050. It can also be solved using SGD as applied to a special loss function called the hinge loss:

$$\text{loss}((\underline{z}_i, y_i); (\underline{x}, v)) = \frac{1}{N} \sum_{i=1}^{N} \max\{1 - y_i(\underline{x}^T \underline{z} + v), 0\},$$

which is a commonly used loss function for classification (you'll learn why in your ML classes).

The reason we are taking this little digression is that applying SGD to the hinge loss gives us The Perceptron Algorithm:

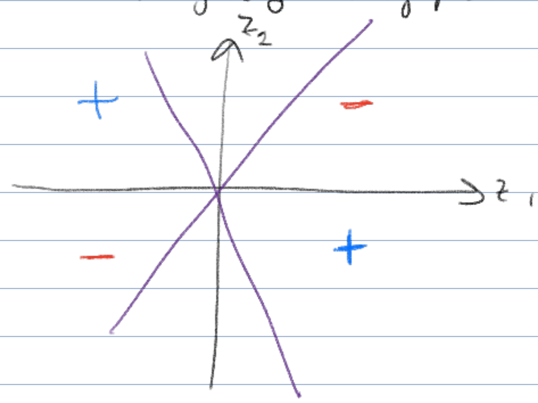
Initialize initial guess $(\underline{x}^{(0)}, v^{0})$
For each iteration $k = 0, 1, 2, \ldots,$ do:
(i) Draw a random index $i \in \{1, \ldots, N\}$
(ii) If $y_i((\underline{x}^{(k)})^T \underline{z}_i + v^{(k)}) < 1$: Update $(\underline{x}^{(k+1)}, v^{(k+1)}) = (\underline{x}^{(k)}, v^{(k)}) + y_i \begin{bmatrix} \underline{z}_i \\ 1 \end{bmatrix}$ (U)
Else: $y_i((\underline{x}^{(k)})^T \underline{z}_i + v^{(k)}) \geq 1$, do not update $(\underline{x}^{(k+1)}, v^{(k+1)}) = (\underline{x}^{(k)}, v^{(k)})$.

This algorithm goes through the examples $(\underline{z}_i, y_i)$ one at a time, and updates the classifier only when it makes a mistake (U). The intuition is it "nudges" the classifier to be "less" wrong by $\|\underline{z}_i\|^2 + 1$ on any example $(\underline{z}_i, y_i)$ it currently misclassifies.

This incremental update works, and you can show that if there exists a solution to (LC), the perceptron algorithm will find it. People got REALLY EXCITED ABOUT THIS. See next page for a NYT article about the perceptron algorithm, which in hindsight seems a little silly given that we now know it's just SGD applied to a particular loss function. But then again, so is most of today's AI!

Single and Multi Layer Perceptrons

Given the excitement about the perceptron, why do we not use them anymore? It turns out, it is very easy to stump! Consider the following set of positive and negative examples:



No linear classifiers can separate the + from the −

Is AI doomed?

These define an XOR function: the positive examples are in quadrants where $\text{sign}(z_1) \neq \text{sign}(z_2)$ and the negative ones are in quadrants for which $\text{sign}(z_1) = \text{sign}(z_2)$. These can't be separated by a linear classifier!

Electronic 'Brain' Teaches Itself

The Navy last week demonstrated the embryo of an electronic computer named the Perceptron which, when completed in about a year, is expected to be the first non-living mechanism able to "perceive, recognize and identify its surroundings without human training or control." Navy officers demonstrating a preliminary form of the device in Washington said they hesitated to call it a machine because it is so much like a "human being without life."

Dr. Frank Rosenblatt, research psychologist at the Cornell Aeronautical Laboratory, Inc., Buffalo, N. Y., designer of the Perceptron. conducted the demonstration. The machine, he said, would be the first electronic device to think as the human brain. Like humans, Perceptron will make mistakes at first, "but it will grow wiser as it gains experience," he said.

The first Perceptron, to cost about $100,000, will have about 1,000 electronic "association cells" receiving electrical impulses from an eyelike scanning device with 400 photocells. The human brain has ten billion responsive cells, including 100,000,000 connections with the eye.

Difference Recognized

The concept of the Perceptron was demonstrated on the Weather Bureau's $2,000,000 IBM 704 computer. In one experiment, the 704 computer was shown 100 squares situated at random either on the left or the right side of a field. In 100 trials, it was able to "say" correctly ninety-seven times whether a square was situated on the right or left. Dr. Rosenblatt said that after having seen only thirty to forty squares the device had learned to recognize the difference between right and left, almost the way a child learns.

When fully developed, the Perceptron will be designed to remember images and information it has perceived itself, whereas ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons, Dr. Rosenblatt said, will be able to recognize people and call out their names. Printed pages, longhand letters and even speech commands are within its reach. Only one more step of development, a difficult step, he said, is needed for the device to hear speech in one language and instantly translate it to speech or writing in another language.

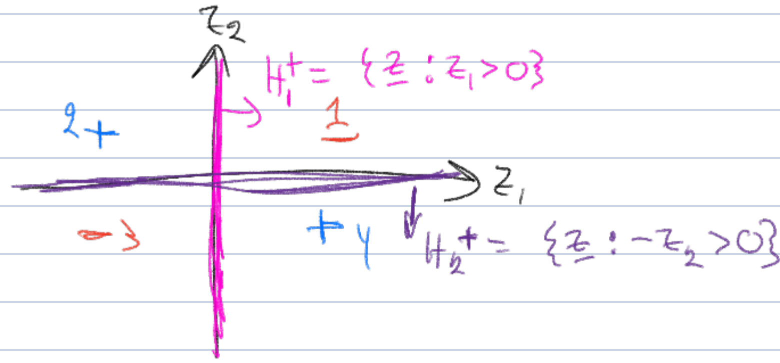
Self-Reproduction

In principle, Dr. Rosenblatt said, it would be possible to build Perceptrons that could reproduce themselves on an assembly line and which would be "conscious" of their existence.

Perceptron, it was pointed out, needs no "priming." It is not necessary to introduce it to surroundings and circumstances, record the data involved and then store them for future comparison as is the case with present "mechanical brains." It literally teaches itself to recognize objects the first time it encounters them. It uses a camera-eye lens to scan objects or survey situations, and an electrical impulse system, patterned point-by-point after the human brain does the interpreting.

The Navy said it would use the principle to build the first Perceptron "thinking machines" that will be able to read or write.

But suppose we were allowed to have two classifiers, and then combine them using a nonlinearity?



In the image above, we define a pink classifier that returns $f_1(\underline{z}) = z_1$ a purple classifier that returns $f_2(\underline{z}) = -z_2$. If we define our output to be

$$f(\underline{z}) = f_1(\underline{z}) f_2(\underline{z}) = -z_1 z_2,$$

then we see that this works!

	1	2	3	4
sign $f_1(\underline{z})$	+1	−1	−1	+1
sign $f_2(\underline{z})$	−1	−1	+1	+1
sign $f(\underline{z})$	−1	+1	−1	+1

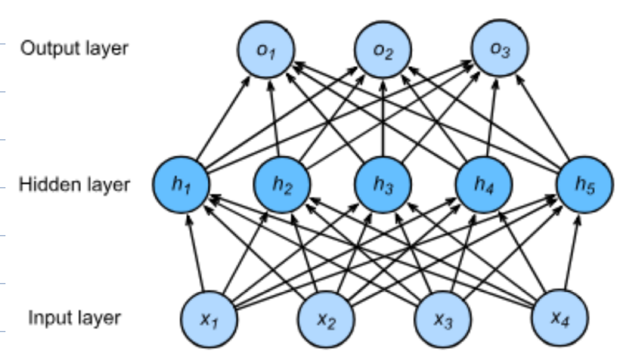
This worked! The two key ingredients here are:

1) Having intermediate computation, called hidden layers
2) Allowing for some nonlinearity.

These two ingredients are combined to define the Multilayer Perceptron (MLP).

A single hidden layer MLP is defined by the equations:

$$\underline{h} = \sigma(W_1 \underline{z} + \underline{b}_1) \quad \text{(MLP1)}$$
$$\underline{o} = W_2 \underline{h} + \underline{b}_2$$



The key features of (MLP1) are:

- An element-wise nonlinearity σ, called an activation function
- The input is $\underline{z} \in \mathbb{R}^n$.
- The hidden layer is defined by a weight matrix $W_1 \in \mathbb{R}^{p_1 \times n}$ and a bias vector $\underline{b}_1 \in \mathbb{R}^{p_1}$
- The output layer is defined by a weight matrix $W_2 \in \mathbb{R}^{p \times p_1}$ and bias vector $\underline{b}_2 \in \mathbb{R}^p$
- The overall map maps input $\underline{z} \in \mathbb{R}^n$ to output $\underline{o} \in \mathbb{R}^p$.

In practice, (MLP1) is trained to find $W_1, W_2, \underline{b}_1, \underline{b}_2$ using SGD and backpropagation.

Why do we need a nonlinear activation function?

Suppose we didn't include $\sigma(\cdot)$, and defined our MLP as $\underline{h} = W_1 \underline{z} + \underline{b}_1$, $\underline{o} = W_2 \underline{h} + \underline{b}_2$.
If we eliminate the hidden layer variable $\underline{h}$, we get

$$\underline{o} = W_2(W_1\underline{z} + \underline{b}_1) + \underline{b}_2 = \underbrace{W_2 W_1}_{W} \underline{z} + \underbrace{W_2 \underline{b}_1 + \underline{b}_2}_{\underline{b}}$$

$$= W\underline{z} + \underline{b}.$$

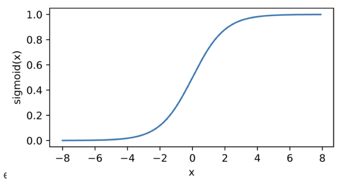
This shows that we do not increase the expressivity of our model, as without the activation function, our model class reduces to affine functions. In some sense, this nonlinearity is the "secret sauce" of MLPs.

Some common activation functions include:

The Sigmoid function:

- Maps input into $(0, 1)$:

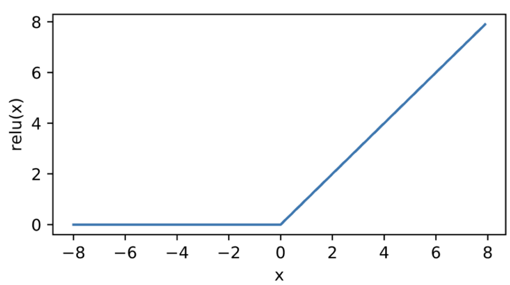
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$



- Can view as a "soft version" of $\sigma(x) = 1$ if $x > 0$ and $\sigma(x) = 0$ if $x \leq 0$.
- This is allows for binary classification over classes $\{0, 1\}$.

The Rectified Linear Unit (ReLU):

$$\text{ReLU}(x) = \max\{x, 0\}.$$

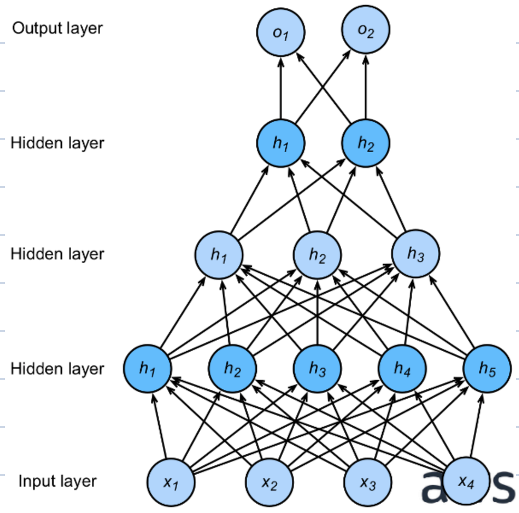


Which activation function to use is a bit of an art, but there are generally accepted tricks of the trade that you'll learn about in ESE 5460. There are also many more than these two, with new ones being invented.

Deep MLPs

There is nothing preventing us from adding more hidden layers! The L-hidden-layer MLP is defined as

$$\underline{h}_1 = \sigma(W_1 \underline{x} + \underline{b}_1)$$
$$\underline{h}_2 = \sigma(W_2 \underline{h}_1 + \underline{b}_2)$$
$$\vdots$$
$$\underline{h}_L = \sigma(W_L \underline{h}_{L-1} + \underline{b}_L)$$
$$\underline{o} = W_{L+1} \underline{h}_L + \underline{b}_{L+1}$$



Shown on the right is an example with 3 hidden layers.

The important thing to notice is these functions are compatible with our discussion on backpropagation, meaning computing gradients with respect to the parameters $W_1, \dots, W_{L+1}, \underline{b}_1, \dots, \underline{b}_{L+1}$ can be done efficiently!

In the online notes, we'll show you how to take advantage of autodifferentiation to efficiently train MLPs in code.