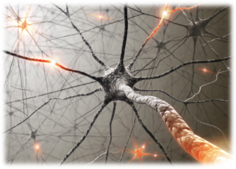
Lecture 3: Machine Learning 2



Roadmap

Stochastic Gradient Descent

Non-linear features

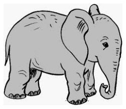
Neural networks

Feature templates

- In this module, we will introduce stochastic gradient descent.

Gradient descent is slow

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



Algorithm: gradient descent

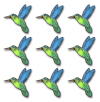
```
Initialize $\mathbf{w} = [0, \dots, 0]$
For $t = 1, \dots, T$:
    $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$
```

Problem: each iteration requires going over all training examples — expensive when have lots of data!

- So far, we've seen gradient descent as a general-purpose algorithm to optimize the training loss.
- But one problem with gradient descent is that it is slow.
- Recall that the training loss is a sum over the training data. If we have one million training examples, then each gradient computation requires going through those one million examples, and this must happen before we can make any progress.
- Can we make progress before seeing all the data?

Stochastic gradient descent

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



Algorithm: stochastic gradient descent

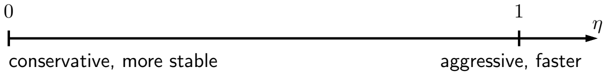
```
Initialize $\mathbf{w} = [0, \dots, 0]$
For $t = 1, \dots, T$:
    For $(x, y) \in \mathcal{D}_{\text{train}}$:
        $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$
```

- The answer is **stochastic gradient descent** (SGD).
- Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples (x, y) and updates the weights $\mathbf{w}$ based on **each** example.
- Each update is not as good because we're only looking at one example rather than all the examples, but we can make many more updates this way.
- Aside: there is a continuum between SGD and GD called minibatch SGD, where each update consists of an average over B examples.
- Aside: There are other variants of SGD. You can randomize the order in which you loop over the training data in each iteration. Think about why this is important if in your training data, you had all the positive examples first and the negative examples after that.

Step size

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Question: what should η be?



Strategies:

- Constant: $\eta = 0.1$
- Decreasing: $\eta = 1/\sqrt{\#\text{ updates made so far}}$

- One remaining issue is choosing the step size, which in practice is quite important.
- Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn.
- On the other hand, with smaller step sizes you get more stability, but you might get to your destination more slowly. Note that the weights do not change if $\eta = 0$
- A suggested form for the step size is to set the initial step size to 1 and let the step size decrease as the inverse of the square root of the number of updates we've taken so far.
- Aside: There are more sophisticated algorithms like AdaGrad and Adam that adapt the step size based on the data, so that you don't have to tweak it as much.
- Aside: There are some nice theoretical results showing that SGD is guaranteed to converge in this case (provided all your gradients are bounded).

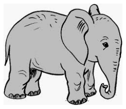
Stochastic gradient descent in Python

[code]

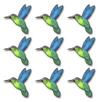
- Now let us code up stochastic gradient descent for linear regression in Python.
- First we generate a large enough dataset so that speed actually matters. We will also generate 1 million points according to $x \sim \mathcal{N}(0, I)$ and $y \sim \mathcal{N}(\mathbf{w}^* \cdot x, 1)$, where $\mathbf{w}^*$ is the true weight vector, but hidden to the algorithm.
- This way, we can diagnose whether the algorithm is actually working or not by checking whether it recovers something close to $\mathbf{w}^*$.
- Let's first run gradient descent, and watch that it makes progress but it is very slow.
- Now let us implement stochastic gradient descent. It is much faster.

Summary

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$



gradient descent



stochastic gradient descent

Key idea: stochastic updates

It's not about **quality**, it's about **quantity**.

- In summary, we've shown how stochastic gradient descent can be faster than gradient descent.
- Gradient just spends too much time refining its gradient (quality), while you can get a quick and dirty estimate just from one sample and make more updates (quantity).
- Of course, sometimes stochastic gradient descent can be unstable, and other techniques such as mini-batching can be used to stabilize it.

Roadmap

Stochastic Gradient Descent

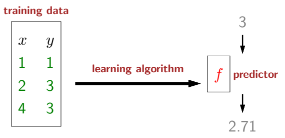
Non-linear features

Neural networks

Feature templates

- In this module, we'll show that even using the machinery of **linear** models, we can obtain much more powerful **non-linear** predictors.

Linear regression



training data

x	y
1	1
2	3
4	3

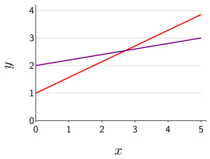
Which predictors are possible?
Hypothesis class

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^d\}$$

$\phi(x) = [1, x]$

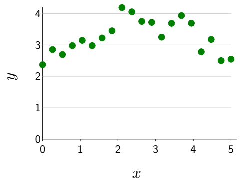
$f(x) = [1, 0.57] \cdot \phi(x)$

$f(x) = [2, 0.2] \cdot \phi(x)$



- We will look at regression and later turn to classification.
- Recall that in linear regression, given training data, a learning algorithm produces a predictor that maps new inputs to new outputs. The first design decision: what are the possible predictors that the learning algorithm can consider (what is the hypothesis class)?
- For linear predictors, remember the hypothesis class is the set of predictors that map some input x to the dot product between some weight vector $\mathbf{w}$ and the feature vector $\phi(x)$.
- As a simple example, if we define the feature extractor to be $\phi(x) = [1, x]$, then we can define various linear predictors with different intercepts and slopes.

More complex data



How do we fit a non-linear predictor?

- But sometimes data might be more complex and not be easily fit by a linear predictor. In this case, what can we do?
- One immediate reaction might be to go to something fancier like neural networks or decision trees.
- But let's see how far we can get with the machinery of linear predictors first.

Quadratic predictors

$$\phi(x) = [1, x, x^2]$$

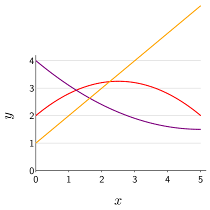
Example: $\phi(3) = [1, 3, 9]$

$f(x) = [2, 1, -0.2] \cdot \phi(x)$

$f(x) = [4, -1, 0.1] \cdot \phi(x)$

$f(x) = [1, 1, 0] \cdot \phi(x)$

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^3\}$



Non-linear predictors just by changing ϕ

- The key observation is that the feature extractor ϕ can be arbitrary.
- So let us define it to include an x^2 term.
- Now, by setting the weights appropriately, we can define a non-linear (specifically, a **quadratic**) predictor.
- The first two examples of quadratic predictors vary in intercept, slope and curvature.
- Note that by setting the weight for feature x^2 to zero, we recover linear predictors.
- Again, the hypothesis class is the set of all predictors $f_{\mathbf{w}}$ obtained by varying $\mathbf{w}$.
- Note that the hypothesis class of quadratic predictors is a **superset** of the hypothesis class of linear predictors.
- In summary, we've seen our first example of obtaining non-linear predictors just by changing the feature extractor ϕ!
- Advanced: here $x \in \mathbb{R}$ is one-dimensional, so x^2 is just one additional feature. If $x \in \mathbb{R}^d$ were d-dimensional, then there would be $O(d^2)$ quadratic features of the form $x_i x_j$ for $i, j \in \{1, \dots, d\}$. When d is large, then d^2 can be prohibitively large, which is one reason that using the machinery of linear predictors to increase expressiveness can be problematic.

Piecewise constant predictors

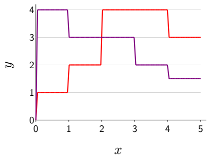
$$\phi(x) = [\mathbf{1}[0 < x \le 1], \mathbf{1}[1 < x \le 2], \mathbf{1}[2 < x \le 3], \mathbf{1}[3 < x \le 4], \mathbf{1}[4 < x \le 5]]$$

Example: $\phi(2.3) = [0, 0, 1, 0, 0]$

$f(x) = [1, 2, 4, 4, 3] \cdot \phi(x)$

$f(x) = [4, 3, 3, 2, 1.5] \cdot \phi(x)$

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^5\}$



Expressive non-linear predictors by partitioning the input space

- Quadratic predictors are still a bit restricted: they can only go up and then down smoothly (or vice-versa).
- We introduce another type of feature extractor which divides the input space into regions and allows the predicted value of each region to vary independently, yielding piecewise constant predictors (see figure).
- Specifically, each component of the feature vector corresponds to one region (e.g., $[0, 1)$) and is 1 if x lies in that region and 0 otherwise.
- Assuming the regions are disjoint, the weight associated with a component/region is exactly the predicted value.
- As you make the regions smaller, then you have more features, and the expressiveness of your hypothesis class increases. In the limit, you can essentially capture any predictor you want.
- Advanced: what happens if x were not a scalar, but a d-dimensional vector? Then if each component gets broken up into B bins, then there will be B^d features! For each feature, we need to fit its weight, and there will in generally be too few examples to fit all the features.

Predictors with periodicity structure

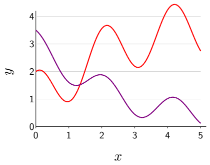
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

Example: $\phi(2) = [1, 2, 4, 0.96]$

$f(x) = [1, 1, -0.1, 1] \cdot \phi(x)$

$f(x) = [3, -1, 0.1, 0.5] \cdot \phi(x)$

$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$



Just throw in any features you want

- Quadratic and piecewise constant predictors are just two examples of an unboundedly large design space of possible feature extractors.
- Generally, the choice of features is informed by the prediction task that we wish to solve (either prior knowledge or preliminary data exploration).
- For example, if x represents time and we believe the true output y varies according to some periodic structure (e.g., traffic patterns repeat daily, sales patterns repeat annually), then we might use periodic features such as cosine to capture these trends.
- Each feature might represent some type of structure in the data. If we have multiple types of structures, these can just be "thrown in" into the feature vector.
- Features represent what properties **might** be useful for prediction. If a feature is not useful, then the learning algorithm can assign a weight close to zero to that feature. Of course, the more features one has, the harder learning becomes.

Linear in what?

Prediction:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

Linear in $\mathbf{w}$? Yes

Linear in $\phi(x)$? Yes

Linear in x? No!

Key idea: non-linearity

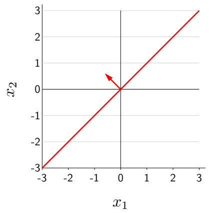
- Expressiveness: score $\mathbf{w} \cdot \phi(x)$ can be a **non-linear** function of x
- Efficiency: score $\mathbf{w} \cdot \phi(x)$ always a **linear** function of $\mathbf{w}$

- Wait a minute...how are we able to obtain non-linear predictors if we're still using the machinery of linear predictors? It's a linguistic sleight of hand, as "linear" is ambiguous.
- The score is $\mathbf{w} \cdot \phi(x)$ linear in $\mathbf{w}$ and $\phi(x)$. However, the score is not linear in x (it might not even make sense because x need not be a vector at all — it could be a string or a PDF file).
- The significance is as follows: From the feature extractor's viewpoint, we can define arbitrary features that yield very **non-linear** functions in x.
- From the learning algorithm's viewpoint (which only looks at $\phi(x)$, not x), **linearity** enables efficient weight optimization.
- Advanced: if the score is linear in $\mathbf{w}$ and the loss function Loss is convex (which holds for the squared, hinge, logistic losses but not the zero-one loss), then minimizing the training loss TrainLoss is a convex optimization problem, and gradient descent with a proper step size is guaranteed to converge to the global minimum.

Linear classification

$\phi(x) = [x_1, x_2]$

$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$



Decision boundary is a line

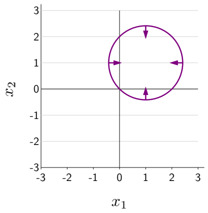
- Now let's turn from regression to classification.
- The story is pretty much the same: you can define arbitrary features to yield non-linear classifiers.
- Recall that in binary classification, the classifier (predictor) returns the sign of the score.
- The classifier can be therefore be represented by its decision boundary, which divides the input space into two regions: points with positive score and points with negative score.
- Note that the classifier $f_{\mathbf{w}}(x)$ is a non-linear function of x (and $\phi(x)$) no matter what (due to the sign function), so it is not helpful to talk about whether $f_{\mathbf{w}}$ is linear or non-linear. Instead we will ask whether the **decision boundary** corresponding to $f_{\mathbf{w}}$ is linear or not.

Quadratic classifiers

$$\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$$
$$f(x) = \text{sign}([2, 2, -1] \cdot \phi(x))$$

Equivalently:

$$f(x) = \begin{cases} 1 & \text{if } \{(x\_1 - 1)^2 + (x\_2 - 1)^2 \le 2\} \\ -1 & \text{otherwise} \end{cases}$$



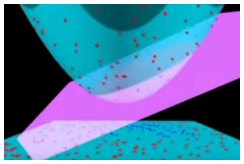
Decision boundary is a circle

- Let us see how we can define a classifier with a non-linear decision boundary.
- Let's try to construct a feature extractor that induces a decision boundary that is a circle: the inside is classified $+1$ and the outside is classified -1.
- We will add a new feature $x_1^2 + x_2^2$ into the feature vector, and define the weights to be as follows.
- Then rewrite the classifier to make it clear that it is the equation for the interior of a circle with radius $\sqrt{2}$.
- As a sanity check, we you can see that $x = [0, 0]$ results in a score of 0, which means that it is on the decision boundary. And as either of x_1 or x_2 grow in magnitude (either $|x_1| \to \infty$ or $|x_2| \to \infty$), the contribution of the third feature dominates and the sign of the score will be negative.

Visualization in feature space

Input space: $x = [x_1, x_2]$, decision boundary is a circle

Feature space: $\phi(x) = [x_1, x_2, x_1^2 + x_2^2]$, decision boundary is a hyperplane



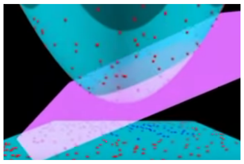
- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$.
- Click on the image to see the linked video (which is about polynomial kernels and SVMs, but the same principle applies here).
- In the input space x, the decision boundary which separates the red and blue points is a circle.
- We can also visualize the points in **feature space**, where each point is given an additional dimension $x_1^2 + x_2^2$.
- In this three-dimensional feature space, a linear predictor (which is now defined by a hyperplane instead of a line) can in fact separate the red and blue points.
- This corresponds to the non-linear predictor in the original two-dimensional space.

Summary

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$

linear in $\mathbf{w}, \phi(x)$

non-linear in x



- Regression: non-linear predictor, classification: non-linear decision boundary
- Types of non-linear features: quadratic, piecewise constant, etc.

Non-linear predictors with linear machinery

- To summarize, we have shown that the term "linear" is ambiguous: a predictor in regression is non-linear in the input x but is linear in the feature vector $\phi(x)$.
- The score is also linear with respect to the weights $\mathbf{w}$, which is important for efficient learning.
- Classification is similar, except we talk about (non-)linearity of the decision boundary.
- We also saw many types of non-linear predictors that you could create by concocting various features (quadratic predictors, piecewise constant predictors).
- So next time someone on the street asks you about linear predictors, you should first ask them "linear in what?"

Roadmap

Stochastic Gradient Descent

Non-linear features

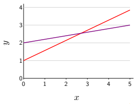
Neural networks

Feature templates

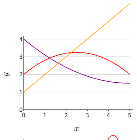
- In this module, I will present neural networks, a way to construct non-linear predictors via problem decomposition.

Non-linear predictors

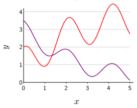
Linear predictors:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x]$



Non-linear (quadratic) predictors:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x),\ \phi(x) = [1, x, x^2]$



Non-linear neural networks:
$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)),\ \phi(x) = [1, x]$



- Recall that our first hypothesis class was linear (in x) predictors, which for regression means that the predictors are lines.
- However, we also showed that you could get non-linear (in x) predictors by simply changing the feature extractor ϕ. For example, by adding the feature x^2, one obtains quadratic predictors.
- One disadvantage of this approach is that if x were d-dimensional, one would need $O(d^2)$ features and corresponding weights, which presents considerable computational and statistical challenges.
- We will show that with neural networks, we can leave the feature extractor alone, but increase the complexity of predictor, which can also produce non-linear (though not necessarily quadratic) predictors.
- It is a common misconception that neural networks allow you to express more complex predictors. You can define ϕ to include essentially all predictors (as is done in kernel methods).
- Rather, neural networks yield non-linear predictors in a more **compact** way. For instance, you might not need $O(d^2)$ features to represent the desired non-linear predictor.

Motivating example

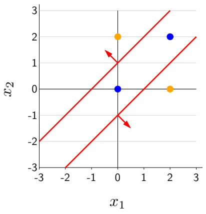
Example: predicting car collision

Input: positions of two oncoming cars $x = [x_1, x_2]$

Output: whether safe ($y = +1$) or collide ($y = -1$)

Unknown: safe if cars sufficiently far: $y = \text{sign}(|x_1 - x_2| - 1)$

x_1	x_2	y
0	2	1
2	0	1
0	0	-1
2	2	-1



- As a motivating example, consider the problem of predicting whether two cars are going to collide given the their positions (as measured from distance from one side of the road). In particular, let x_1 be the position of one car and x_2 be the position of the other car.
- Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. This relationship can be represented by the decision boundary which labels all points in the interior region between the two red lines as negative, and everything on the exterior (on either side) as positive. Of course, this true input-output relationship is unknown to the learning algorithm, which only sees training data. Consider a simple training dataset consisting of four points. (This is essentially the famous XOR problem that was impossible to fit using linear classifiers.)

Decomposing the problem

Test if car 1 is far right of car 2:

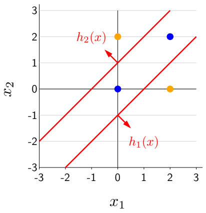
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$f(x) = \text{sign}(h_1(x) + h_2(x))$$



x	$h_1(x)$	$h_2(x)$	$f(x)$
$[0,2]$	0	1	$+1$
$[2,0]$	1	0	$+1$
$[0,0]$	0	0	-1
$[2,2]$	0	0	-1

- One way to motivate neural networks (without appealing to the brain) is **problem decomposition**.
- The intuition is to break up the full problem into two subproblems: the first subproblem tests if car 1 is to the far right of car 2; the second subproblem tests if car 2 is to the far right of car 1. Then the final output is 1 iff at least one of the two subproblems returns 1.
- Concretely, we can define $h_1(x)$ to be the output of the first subproblem, which is a simple linear decision boundary (in fact, the right line in the figure).
- Analogously, we define $h_2(x)$ to be the output of the second subproblem.
- Note that $h_1(x)$ and $h_2(x)$ take on values 0 or 1 instead of -1 or $+1$.
- The points can then be classified by first computing $h_1(x)$ and $h_2(x)$, and then combining the results into $f(x)$.

Rewriting using vector notation

Intermediate subproblems:

$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1] = \mathbf{1}[[-1,+1,-1] \cdot [1,x_1,x_2] \geq 0]$$

$$h_2(x) = \mathbf{1}[x_2 - x_1 \geq 1] = \mathbf{1}[[-1,-1,+1] \cdot [1,x_1,x_2] \geq 0]$$

$$\mathbf{h}(x) = \mathbf{1}\left[\begin{bmatrix} -1 & +1 & -1 \\ -1 & -1 & +1 \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \geq 0\right]$$

Predictor:

$$f(x) = \text{sign}(h_1(x) + h_2(x)) = \text{sign}([1,1] \cdot \mathbf{h}(x))$$

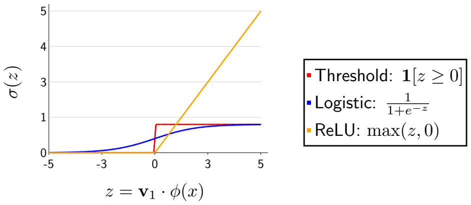
- Now let us rewrite this predictor $f(x)$ using vector notation.
- We can define a feature vector $[1, x_1, x_2]$ and a corresponding weight vector, where the dot product thresholded yields exactly $h_1(x)$.
- We do the same for $h_2(x)$.
- We put the two subproblems into one equation by stacking the weight vectors into one matrix. Recall that left-multiplication by a matrix is equivalent to taking the dot product with each row. By convention, the thresholding at 0 ($\mathbf{1}[\cdot \geq 0]$) applies component-wise.
- Finally, we can define the predictor in terms of a simple dot product.
- Now of course, we don't know the weight vectors, but we can learn them from the training data!

Avoid zero gradients

Problem: gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is 0

$$h_1(x) = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

Solution: replace with an **activation function** σ with non-zero gradients

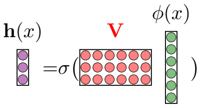


$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

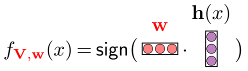
- Later we'll show how to perform learning using gradient descent, but we can anticipate one problem, which we encountered when we tried to optimize the zero-one loss.
- The gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is always zero because of the threshold function.
- To fix this, we replace the threshold function with an **activation function** with non-zero gradients
- Classically, neural networks used the **logistic function** $\sigma(z)$, which looks roughly like the threshold function but has non-zero gradients everywhere.
- Even though the gradients are non-zero, they can be quite small when $|z|$ is large (a phenomenon known as saturation). This makes optimizing with the logistic function still difficult.
- In 2012, Glorot et al. introduced the ReLU activation function, which is simply $\max(z, 0)$. This has the advantage that at least on the positive side, the gradient does not vanish (though on the negative side, the gradient is always zero). As a bonus, ReLU is easier to compute (only max, no exponentiation). In practice, ReLU works well and has become the activation function of choice.
- Note that if the activation function were linear (e.g., the identity function), then the gradients would always be nonzero, but you would lose the power of a neural network, because you would simply get the product of the final-layer weight vector and the weight matrix ($\mathbf{w}^\top \mathbf{V}$), which is equivalent to optimizing over a single weight vector.
- Therefore, that there is a tension between wanting an activation function that is non-linear but also has non-zero gradients.

Two-layer neural networks

Intermediate subproblems:



Predictor (classification):



Interpret $\mathbf{h}(x)$ as a learned feature representation!

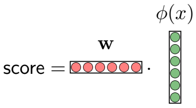
Hypothesis class:

$$\mathcal{F} = \{f_{\mathbf{V},\mathbf{w}} : \mathbf{V} \in \mathbb{R}^{k \times d}, \mathbf{w} \in \mathbb{R}^k\}$$

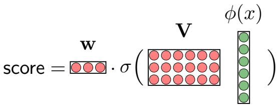
- Now we are finally ready to define the hypothesis class of two-layer neural networks.
- We start with a feature vector $\phi(x)$.
- We multiply it by a weight matrix $\mathbf{V}$ (whose rows can be interpreted as the weight vectors of the k intermediate subproblems.
- Then we apply the activation function σ to each of the k components to get the hidden representation $\mathbf{h}(x) \in \mathbb{R}^k$.
- We can actually interpret $\mathbf{h}(x)$ as a learned feature vector (representation), which is derived from the original non-linear feature vector $\phi(x)$.
- Given $\mathbf{h}(x)$, we take the dot product with a weight vector $\mathbf{w}$ to get the score used to drive either regression or classification.
- The hypothesis class is the set of all such predictors obtained by varying the first-layer weight matrix $\mathbf{V}$ and the second-layer weight vector $\mathbf{w}$.

Deep neural networks

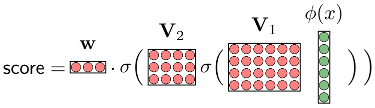
1-layer neural network:



2-layer neural network:



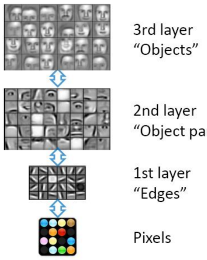
3-layer neural network:



- We can push these ideas to build deep neural networks, which are neural networks with many layers.
- Warm up: for a one-layer neural network (a.k.a. a linear predictor), the score that drives prediction is simply a dot product between a weight vector and a feature vector.
- We just saw for a two-layer neural network, we apply a linear layer $\mathbf{V}$ first, followed by a non-linearity σ, and then take the dot product.
- To obtain a three-layer neural network, we apply a linear layer and a non-linearity (this is the basic building block). This can be iterated any number of times. No matter now deep the neural network is, the top layer is always a linear function, and all the layers below that can be interpreted as defining a (possibly very complex) hidden feature vector.
- In practice, you would also have a bias term (e.g., $\mathbf{V}\phi(x) + b$). We have omitted all bias terms for notational simplicity.

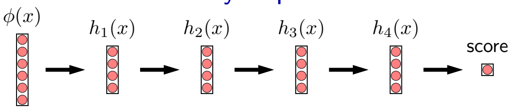
[figure from Honglak Lee]

Layers represent multiple levels of abstractions



- It can be difficult to understand what a sequence of (matrix multiply, non-linearity) operations buys you.
- To provide intuition, suppose the input feature vector $\phi(x)$ is a vector of all the pixels in an image.
- Then each layer can be thought of producing an increasingly abstract representation of the input. The first layer detects edges, the second detects object parts, the third detects objects. What is shown in the figure is for each component j of the hidden representation $\mathbf{h}(x)$, the input image $\phi(x)$ that maximizes the value of $h_j(x)$.
- Though we haven't talked about learning neural networks, it turns out that the "levels of abstraction" story is actually borne out visually when we learn neural networks on real data (e.g., images).

Why depth?

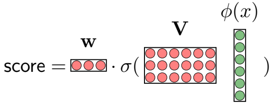


Intuitions:

- Multiple levels of abstraction
- Multiple steps of computation
- Empirically works well
- Theory is still incomplete

- Beyond learning hierarchical feature representations, deep neural networks can be interpreted in a few other ways.
- One perspective is that each layer can be thought of as performing some computation, and therefore deep neural networks can be thought of as performing multiple steps of computation.
- But ultimately, the real reason why deep neural networks are interesting is because they work well in practice.
- From a theoretical perspective, we have a quite an incomplete explanation for why depth is important. The original motivation from McCulloch/Pitts in 1943 showed that neural networks can be used to simulate a bounded computation logic circuit. Separately it has been shown that depth $k+1$ logic circuits can represent more functions than depth k. However, neural networks are real-valued and might have types of computations which don't fit neatly into logical paradigm. Obtaining a better theoretical understanding is an active area of research in statistical learning theory.

Summary



- Intuition: decompose problem into intermediate parallel subproblems
- Deep networks iterate this decomposition multiple times
- Hypothesis class contains predictors ranging over weights for all layers

- To summarize, we started with a toy problem (the XOR problem) and used it to motivate neural networks, which decompose a problem into intermediate subproblems, which are solved in parallel.
- Deep networks iterate this multiple times to build increasingly high-level representations of the input.

Roadmap

Stochastic Gradient Descent

Non-linear features

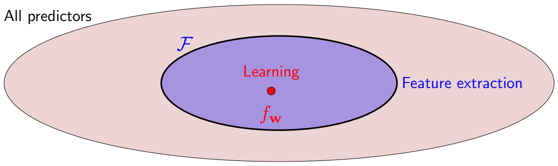
Neural networks

Feature templates

- In this module, we'll talk about how to use feature templates to construct features in a flexible way.

Feature extraction + learning

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$



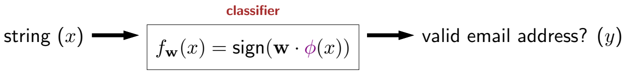
- Feature extraction: choose $\mathcal{F}$ based on domain knowledge
- Learning: choose $f_{\mathbf{w}} \in \mathcal{F}$ based on data

Want $\mathcal{F}$ to contain good predictors but not be too big

- Recall that the hypothesis class $\mathcal{F}$ is the set of predictors considered by the learning algorithm. In the case of linear predictors, $\mathcal{F}$ is given by some function of $\mathbf{w} \cdot \phi(x)$ for all $\mathbf{w}$ (sign for classification, no sign for regression). This can be visualized as a set in the figure.
- Learning is the process of choosing a particular predictor $f_{\mathbf{w}}$ from $\mathcal{F}$ given training data.
- But the question that will concern us in this module is how do we choose $\mathcal{F}$? We saw some options already: linear predictors, quadratic predictors, etc., but what makes sense for a given application?
- If the hypothesis class doesn't contain any good predictors, then no amount of learning can help. So the question when extracting features is really whether they are powerful enough to **express** good predictors. It's okay and expected that $\mathcal{F}$ will contain bad ones as well. Of course, we don't want $\mathcal{F}$ to be too big, or else learning becomes hard, not just computationally but statistically (as we'll explain when we talk about generalization).

Feature extraction with feature names

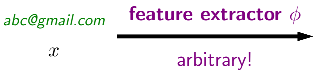
Example task:



string (x) → $f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$ → valid email address? (y)

Question: what properties of x **might be** relevant for predicting y?

Feature extractor: Given x, produce set of (feature name, feature value) pairs



length>10	: 1
fracOfAlpha	: 0.85
contains_@	: 1
endsWith_com	: 1
endsWith_org	: 0

$\phi(x)$

- To get some intuition about feature extraction, let us consider the task of predicting whether whether a string is a valid email address or not.
- We will assume the classifier $f_{\mathbf{w}}$ is a linear classifier, which is given by some feature extractor ϕ.
- Feature extraction is a bit of an art that requires intuition about both the task and also what machine learning algorithms are capable of. The general principle is that features should represent properties of x which **might be** relevant for predicting y.
- Think about the feature extractor as producing a set of (feature name, feature value) pairs. For example, we might extract information about the length, or fraction of alphanumeric characters, whether it contains various substrings, etc.
- It is okay to add features which turn out to be irrelevant, since the learning algorithm can always in principle choose to ignore the feature, though it might take more data to do so.
- We have been associating each feature with a name so that it's easier for us (humans) to interpret and develop the feature extractor. The feature names act like the analogue of **comments** in code. Mathematically, the feature name is not needed by the learning algorithm and erasing them does not change prediction or learning.

Prediction with feature names

Weight vector $\mathbf{w} \in \mathbb{R}^d$

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith_com	:2.2
endsWith_org	:1.4

Feature vector $\phi(x) \in \mathbb{R}^d$

length>10	:1
fracOfAlpha	:0.85
contains_@	:1
endsWith_com	:1
endsWith_org	:0

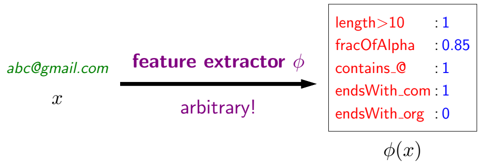
Score: weighted combination of features

$$\mathbf{w} \cdot \phi(x) = \sum_{j=1}^{d} w_j \phi(x)_j$$

Example: $-1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51$

- A feature vector formally is just a list of numbers, but we have endowed each feature in the feature vector with a name.
- The weight vector is also just a list of numbers, but we can endow each weight with the corresponding name as well.
- Recall that the score is simply the dot product between the weight vector and the feature vector. In other words, the score aggregates the contribution of each feature, weighted appropriately.
- Each feature weight w_j determines how the corresponding feature value $\phi_j(x)$ contributes to the prediction.
- If w_j is positive, then the presence of feature j ($\phi_j(x) = 1$) favors a positive classification (e.g., ending with com). Conversely, if w_j is negative, then the presence of feature j favors a negative classification (e.g., length greater than 10). The magnitude of w_j measures the strength or importance of this contribution.
- Advanced: while tempting, it can be a bit misleading to interpret feature weights in isolation, because the learning algorithm treats $\mathbf{w}$ holistically. In particular, a feature weight w_j produced by a learning algorithm will change depending on the presence of other features. If the weight of a feature is positive, it doesn't necessarily mean that feature is positively correlated with the label.

Organization of features?



abc@gmail.com

x

feature extractor ϕ

arbitrary!

length>10 : 1
fracOfAlpha : 0.85
contains_@ : 1
endsWith_com : 1
endsWith_org : 0

$\phi(x)$

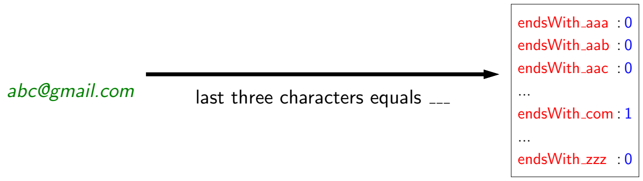
Which features to include? Need an organizational principle...

- How would we go about about creating good features?
- Here, we used our prior knowledge to define certain features (contains_@) which we believe are helpful for detecting email addresses.
- But this is ad-hoc, and it's easy to miss useful features (e.g., endsWith_us), and there might be other features which are predictive but not intuitive.
- We need a more systematic way to go about this.

Feature templates

Definition: feature template

A **feature template** is a group of features all computed in a similar way.



abc@gmail.com

last three characters equals ___

endsWith_aaa : 0
endsWith_aab : 0
endsWith_aac : 0
...
endsWith_com : 1
...
endsWith_zzz : 0

Define types of pattern to look for, not particular patterns

- A useful organization principle is a **feature template**, which groups all the features which are computed in a similar way. (People often use the word "feature" when they really mean "feature template".)
- Rather than defining individual features like endsWith_com, we can define a single feature template which expands into all the features that computes whether the input x matches any three characters.
- Typically, we will write a feature template as an English description with a blank (__), which is to be filled in with an arbitrary value.
- The upshot is that we don't need to know which particular patterns (e.g., three-character suffixes) are useful, but only that **existence** of certain patterns (e.g., three-character suffixes) are useful cue to look at.
- It is then up to the learning algorithm to figure out which patterns are useful by assigning the appropriate feature weights.

Feature templates example 1

Input:

abc@gmail.com

Feature template	Example feature		
Last three characters equals ___	Last three characters equals *com*	:	1
Length greater than ___	Length greater than *10*	:	1
Fraction of alphanumeric characters	Fraction of alphanumeric characters	:	0.85

- Here are some other examples of feature templates.
- Note that an isolated feature (e.g., fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- In many cases, the feature value is binary (0 or 1), but they can also be real numbers.

Feature templates example 2

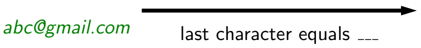
Input:

Latitude: 37.4068176
Longitude: -122.1715122

Feature template	Example feature name	
Pixel intensity of image at row ___ and column ___ (___ channel)	Pixel intensity of image at row *10* and column *93* (*red* channel)	: 0.8
Latitude is in [___, ___] and longitude is in [___, ___]	Latitude is in [*37.4*, *37.5*] and longitude is in [*-122.2*, *-122.1*]	: 1

- As another example application, suppose the input is an aerial image along with the latitude/longitude corresponding where the image was taken. This type of input arises in poverty mapping and land cover classification.
- In this case, we might define one feature template corresponding to the pixel intensities at various pixel-wise row/column positions in the image across all the 3 color channels (e.g., red, green, blue).
- Another feature template might define a family of binary features, one for each region of the world, where each region is defined by a bounding box over latitude and longitude.

Sparsity in feature vectors



endsWith_a	: 0
endsWith_b	: 0
endsWith_c	: 0
endsWith_d	: 0
endsWith_e	: 0
endsWith_f	: 0
endsWith_g	: 0
endsWith_h	: 0
endsWith_i	: 0
endsWith_j	: 0
endsWith_k	: 0
endsWith_l	: 0
endsWith_m	: **1**
endsWith_n	: 0
endsWith_o	: 0
endsWith_p	: 0
endsWith_q	: 0
endsWith_r	: 0
endsWith_s	: 0
endsWith_t	: 0
endsWith_u	: 0
endsWith_v	: 0
endsWith_w	: 0
endsWith_x	: 0
endsWith_y	: 0
endsWith_z	: 0

Compact representation:

```
{"endsWith_m": 1}
```

- In general, a feature template corresponds to many features, and sometimes, **for a given input**, most of the feature values are zero; that is, the feature vector is **sparse**.
- Of course, different feature vectors have different non-zero features.
- In this case, it would be inefficient to represent all the features explicitly. Instead, we can just store the values of the non-zero features, assuming all other feature values are zero by default.

Two feature vector implementations

Arrays (good for dense features):

pixelIntensity(0,0)	: 0.8
pixelIntensity(0,1)	: 0.6
pixelIntensity(0,2)	: 0.5
pixelIntensity(1,0)	: 0.5
pixelIntensity(1,1)	: 0.8
pixelIntensity(1,2)	: 0.7
pixelIntensity(2,0)	: 0.2
pixelIntensity(2,1)	: 0
pixelIntensity(2,2)	: 0.1

```
[0.8, 0.6, 0.5, 0.5, 0.8, 0.7, 0.2, 0, 0.1]
```

Dictionaries (good for sparse features):

fracOfAlpha	: 0.85
contains_a	: 0
contains_b	: 0
contains_c	: 0
contains_d	: 0
contains_e	: 0
...	
contains_@	: 1
...	

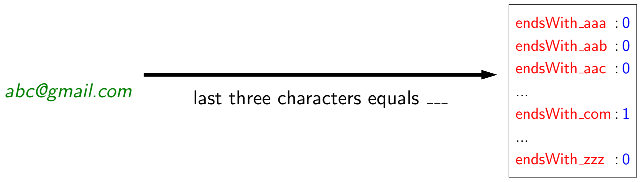
```
{"fracOfAlpha": 0.85, "contains_@": 1}
```

- In general, there are two common ways to implement feature vectors: using arrays and using dictionaries.
- **Arrays** assume a fixed ordering of the features and store the feature values as an array. This implementation is appropriate when the number of nonzeros is significant (the features are dense). Arrays are especially efficient in terms of space and speed (and you can take advantage of GPUs). In computer vision applications, features (e.g., the pixel intensity features) are generally dense, so arrays are more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to implement the feature vector as a **dictionary** (map) from strings to doubles rather than a fixed-size array of doubles. The features not in the dictionary implicitly have a default value of zero. This sparse implementation is useful for natural language processing with linear predictors, and is what allows us to work efficiently over millions of features. In Python, one would define a feature vector $\phi(x)$ as the dictionary `{"endsWith_"+x[-3:]: 1}`. Dictionaries do incur extra overhead compared to arrays, and therefore dictionaries are much slower when the features are not sparse.
- One advantage of the sparse feature implementation is that you don't have to instantiate all the set of possible features in advance; the weight vector can be initialized to {}, and only when a feature weight becomes non-zero do we store it. This means we can dynamically update a model with incrementally arriving data, which might instantiate new features.

Summary

$$\mathcal{F} = \{f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) : \mathbf{w} \in \mathbb{R}^d\}$$

Feature template:



Dictionary implementation:

{"endsWith_com": 1}

- The question we are concerned with in this module is to how to define the hypothesis class $\mathcal{F}$, which in the case of linear predictors is the question of what the feature extractor ϕ is.
- We showed how **feature templates** can be useful for organizing the definition of many features, and that we can use dictionaries to represent **sparse** feature vectors efficiently.
- Stepping back, feature engineering is one of the most critical components in the practice of machine learning. It often does not get as much attention as it deserves, mostly because it is a bit of an art and somewhat domain-specific.
- More powerful predictors such as neural networks will alleviate some of the burden of feature engineering, but even neural networks use feature vectors as the initial starting point, and therefore its effectiveness is ultimately governed by how good the features are.

Overall Summary

- Stochastic Gradient Descent: faster gradient descent using sample gradients
- Non-Linear Features: Linear in weights $\mathbf{w}$, but nonlinear in inputs x
- Neural networks: Learning hierarchical feature representations
- Feature templates: useful for organizing the definition of many features,
- Next: Backpropagation, k-means, generalization, best practices

- In summary, we started with stochastic gradient descent which can be faster than gradient descent. with the cost of noisier updates
- Then, we discussed non-linear features, and outlines a general recipe for linear models that are nonlinear in the original inputs by using the feature vector mapping
- Then, we discussed neural networks, which can be interpreted as an parametric approach for learning flexible hierarchical feature representations
- Finally, we covered how feature templates can be useful for organizing the definition of many features
- Next Lecture, we will cover backpropagation, briefly cover kmeans for unsupervised learning, then discuss generalization and best practices