Roadmap

Topics in the lecture:

Nonlinear features

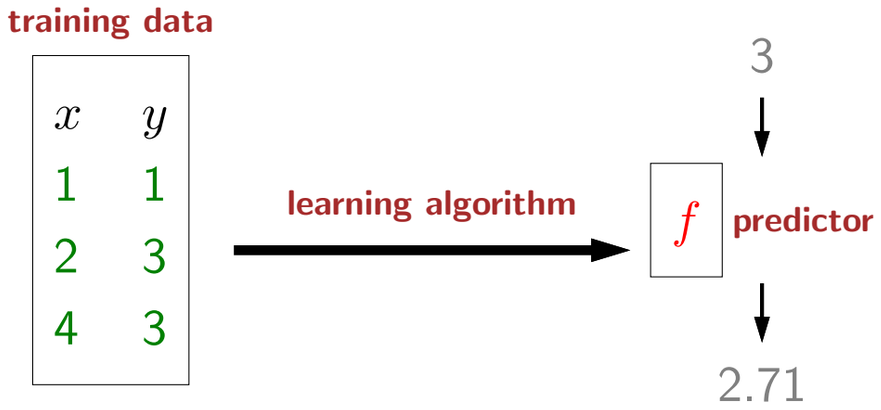
Feature templates

Neural networks

Backpropagation

- In this lecture, we are going to cover four topics: non-linear features, which are ways to use linear models to create nonlinear predictors.
- We will then follow this up with feature templates, which are flexible ways of defining features
- These two topics tell us how to hand-craft features to make linear models more effective
- We will then follow this up with neural nets, which give the possibility of automatically learning these features from data
- Training these can be difficult, and we will end by discussing backpropataion, which makes this possible

Linear regression



Which predictors are possible?

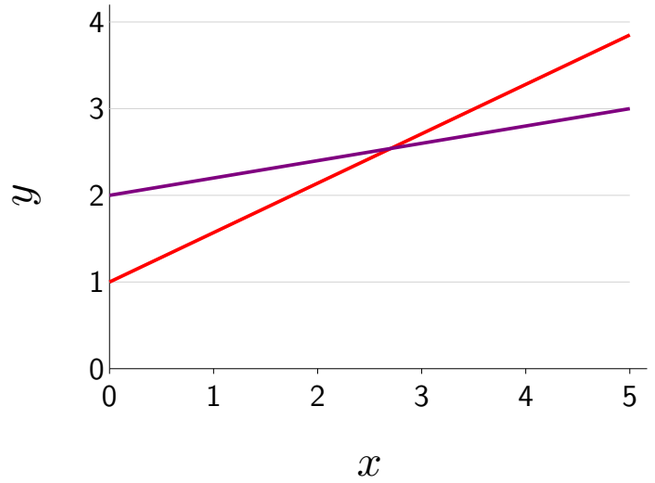
Hypothesis class

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

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

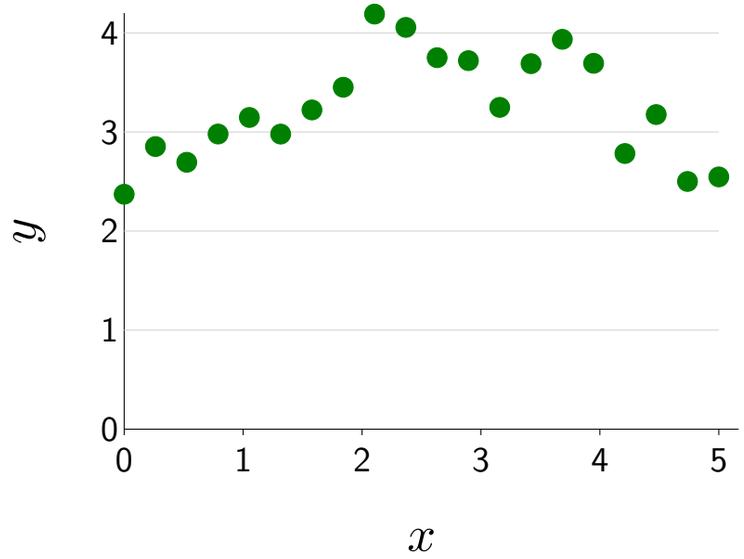
$f(x) = [1, 0.57] \cdot \phi(x)$

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



- We will look at regression and later turn to classification.
- Last week we defined linear regression as a procedure which takes training data and produces a predictor that maps new inputs to new outputs.
- We discussed three parts to this problem, and the first one was the hypothesis class. This is the set of possible predictors for the learning problem
- For linear predictors, 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

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

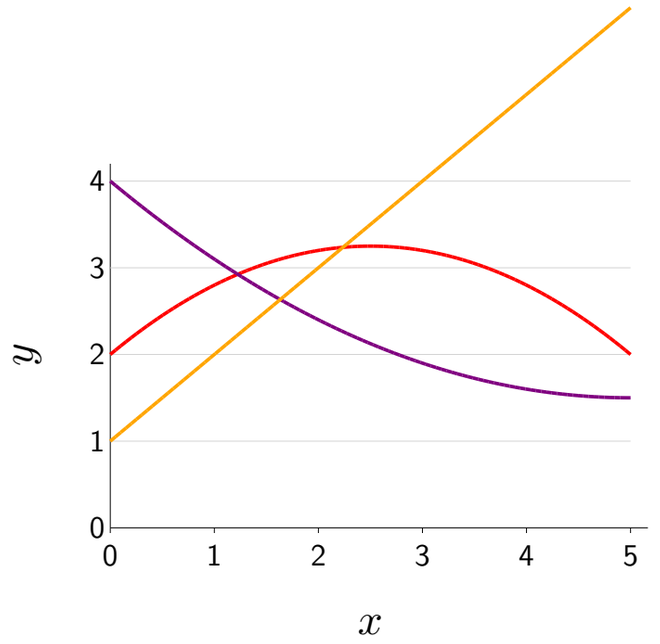
Example: $\phi(3) = [1, 3, 9]$

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

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

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

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



Non-linear predictors just by changing ϕ

- A linear model takes some input and makes a prediction that is linear in its inputs. But there is no rule saying that we cannot non-linearly transform the input data first.
- This is going to be the key starting observation – the feature extractor ϕ can be arbitrary
- As an example, if we wanted to fit a quadratic function, we can augment ϕ by a x^2 term.
- Now, by setting the weights appropriately, we can define a non-linear (specifically, a **quadratic**) predictor.
- 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: try thinking about how to use this idea to learn two- or even three- dimensional quadratics. What new challenges arise?

Piecewise constant predictors

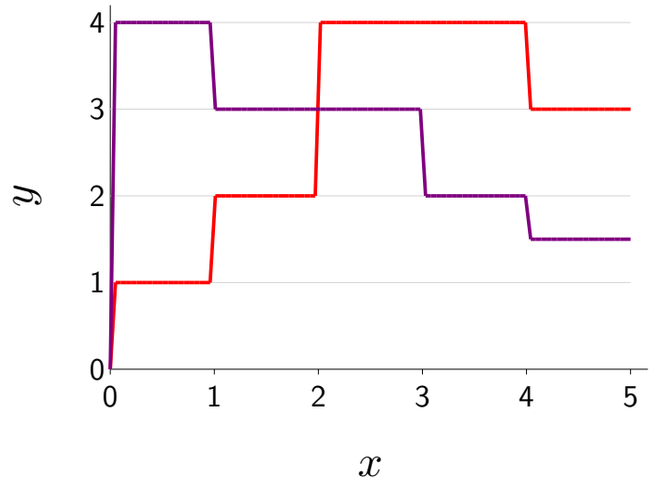
$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]$$

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-vera).
- 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 expressivity of your hypothesis class increases. In the limit, you can essentially capture any predictor you want.
- The drawback is that you need alot of training data to learn these predictors – doing this trick in d-dimensions, your sample requirements go up exponentially in the dimension!

Predictors with periodicity structure

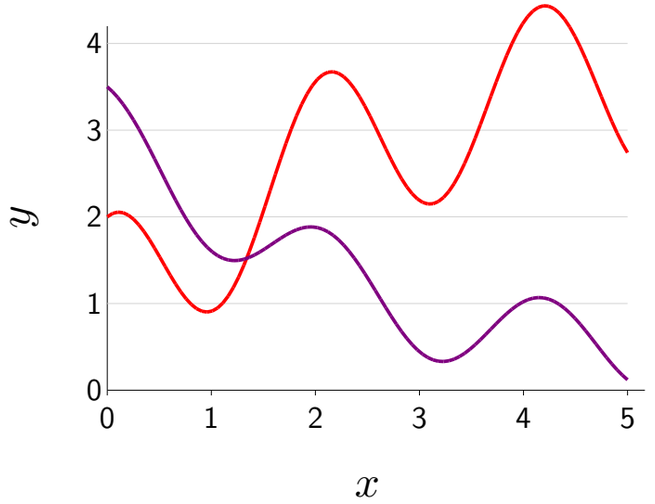
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

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

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

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

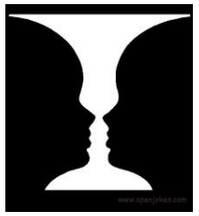
$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$



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!

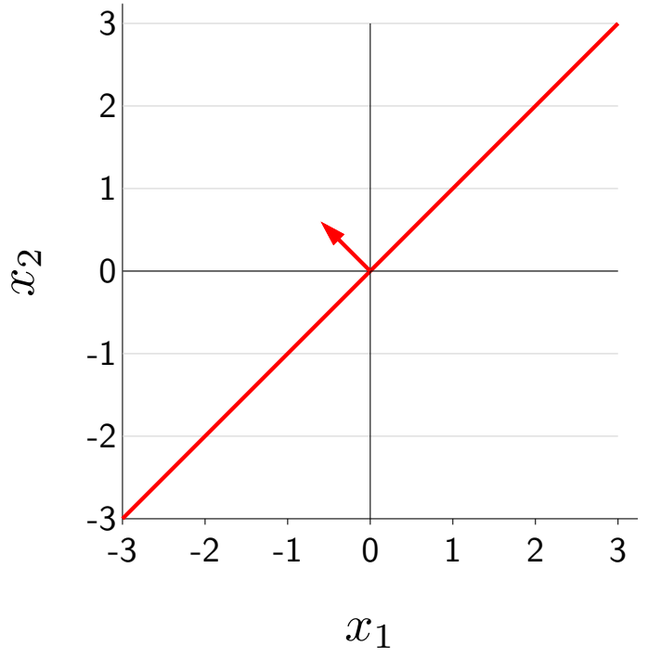
Key idea: non-linearity

- Expressivity: 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? Think of this like pre-processing: the feature map phi is a way of transforming the data so that it is linear
- The score $\mathbf{w} \cdot \phi(x)$ is 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. In NLP, x is often a piece of text).
- Adding more features will generally make the prediction problem more linear, but this comes at the cost of learning. We have to learn more weights in the linear regression, which will require larger datasets
- The machinery of non-linear features combines the benefits of linear models – like its simplicity and ease of optimization, with those of more complex models that capture nonlinear relations.

Linear classification

$$\phi(x) = [x_1, x_2]$$
$$f(x) = \text{sign}([-0.6, 0.6] \cdot \phi(x))$$



Decision boundary is a line

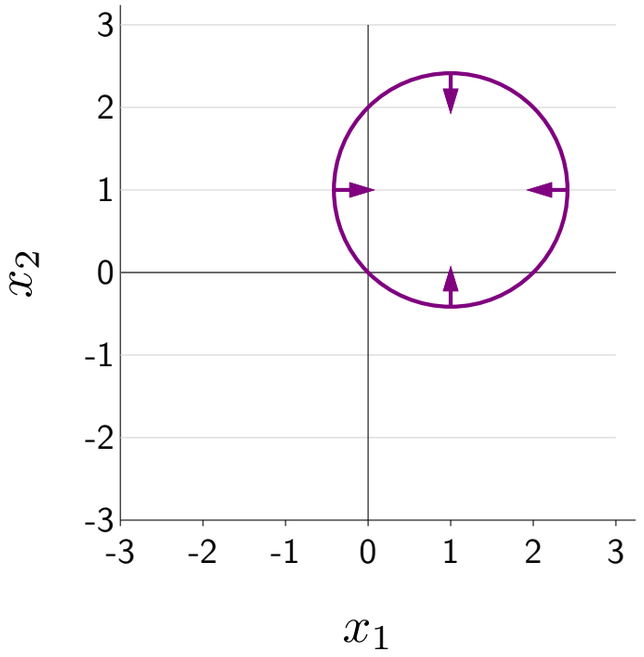
- Now let's turn from regression to classification.
- The story is pretty much the same: you can define arbitrary features to yield non-linear classifiers.
- Recall that in binary classification, the classifier (predictor) returns the sign of the score.
- The classifier can be therefore be represented 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:

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



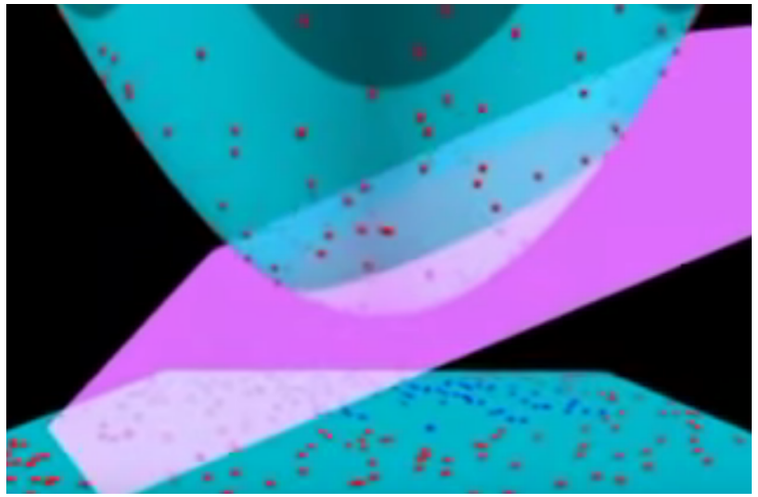
Decision boundary is a circle

- 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



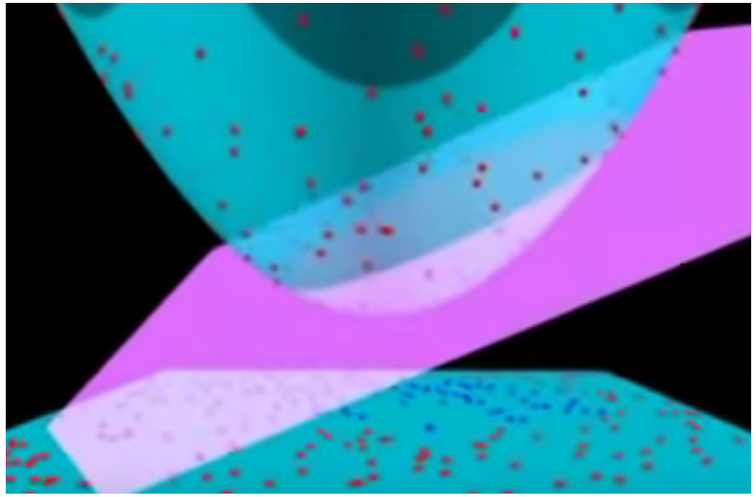
- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$.
- Click on the image to see the linked video (which is about polynomial kernels and SVMs, but the same principle applies here).
- In the input space x, the decision boundary which separates the red and 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).
- The takeaway is that linear models are surprisingly powerful! More sophisticated versions of this called kernels drove much of the statistical machine learning gains in the early 2000s!

Roadmap

Topics in the lecture:

Nonlinear features

Feature templates

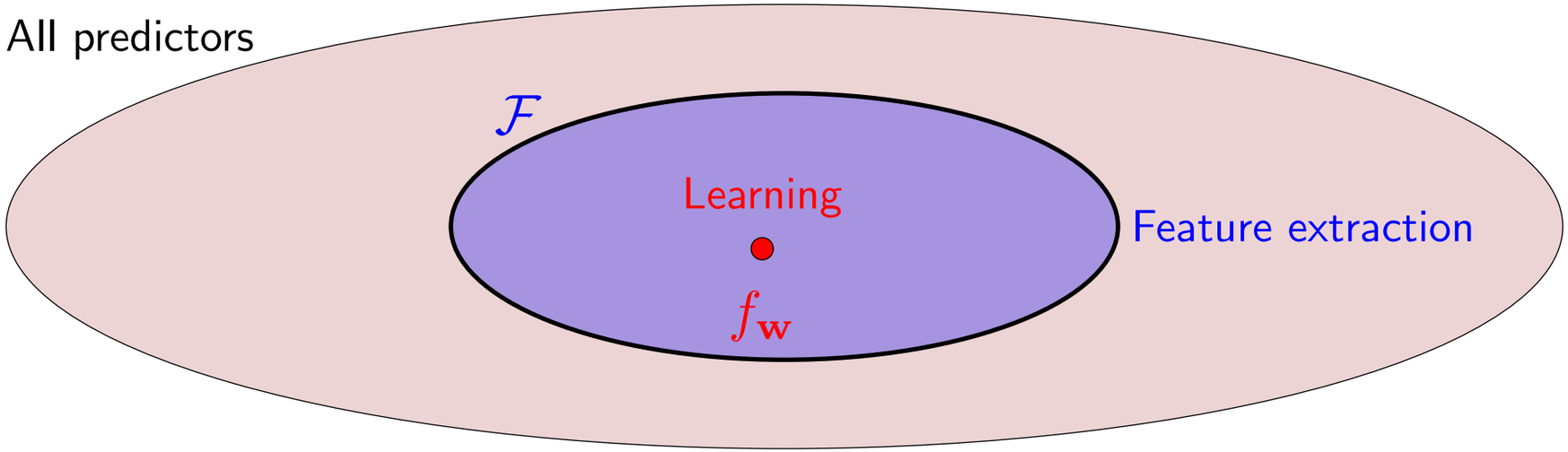
Neural networks

Backpropagation

- Hopefully, you now have an idea of how to use linear models to learn non-linear predictors
- The main challenge though, is in defining these non-linear features.
- We will now cover feature templates, which is a way of defining flexible families of features

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 lecture 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:

classifier

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

abc@gmail.com

x

feature extractor ϕ

arbitrary!

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

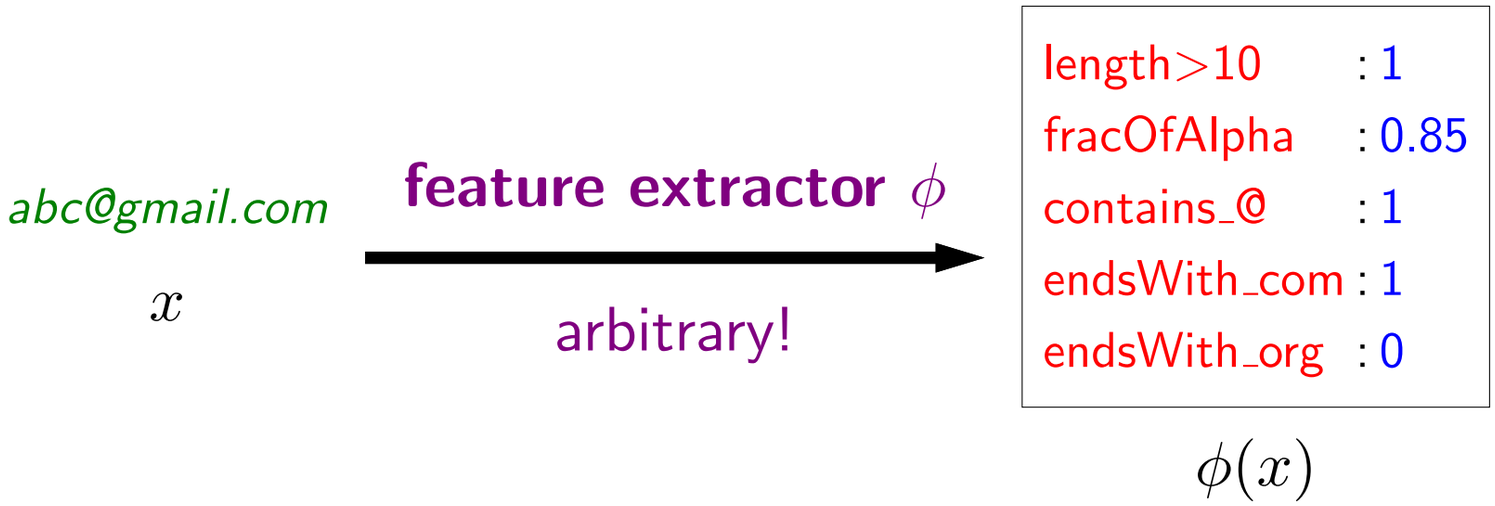
Score: weighted combination of features

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

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

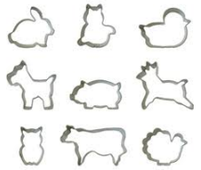
- A feature vector formally is just a list of numbers, but we have endowed each feature in the feature vector with a name.
- The weight vector is also just a list of numbers, but we can endow each weight with the corresponding name as well.
- Recall that the score is simply the dot product between the 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.
- As a side note - it might be tempting to try to interpret these weights as correlations. Does a positive weight mean that a feature is positively correlated with the output? Not necessarily - the weights in a model work together, and a feature with positive weight can even have negative correlation with the outcome. You may have seen an example of this in a stats course - called simpsons paradox.

Organization of features?

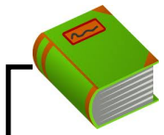


Which features to include? Need an organizational principle...

- How would we go about about creating good features?
- Here, we used our prior knowledge to define certain features (like whether an input contains_@) which we believe are helpful for detecting email addresses.
- But this is ad-hoc, and it's easy to miss useful features, 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.

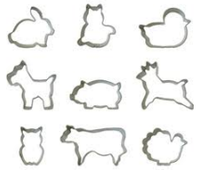
abc@gmail.com

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.



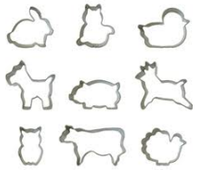
Feature templates example 1

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

Pixel intensity of image at row ___ and column ___ (___ channel)

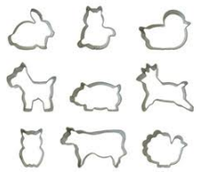
Latitude is in [___, ___] and longitude is in [___, ___]

Example feature name

Pixel intensity of image at row *10* and column *93* (*red* channel) : 0.8

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 to 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

abc@gmail.com

last character equals ___

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. 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 empty $\{\}$, 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:

abc@gmail.com

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

Dictionary implementation:

```
{"endsWith_com":  1}
```

- The question we are concerned with in this section is 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.

Roadmap

Topics in the lecture:

Nonlinear features

Feature templates

Neural networks

Backpropagation

- Thus far, we have been making linear models more and more flexible by designing complex features
- But this is time-consuming, and its often hard for humans to write down the right features
- Neural nets have the promise of making this process automatic - neural nets can be thought of as automated feature extractors
- We will begin by defining and understanding neural networks as a model
- Afterwards, we will discuss how to compute gradients for these models using backpropagation

What is this animal?

- What is the right feature extractor for a zebra?

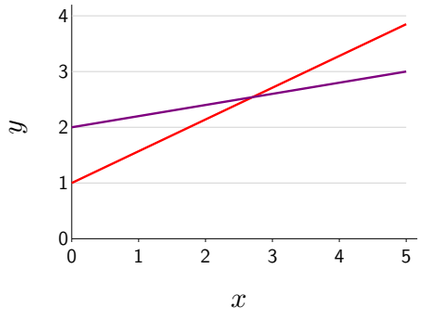
- The goal: automatically learn a feature extractor

- Lets return to our running example of building a zebra detector
- Whats the right set of features here? its honestly pretty hard! maybe you use a set of hand-written texture matchers
- But this seems error prone. We want our model to learn whats a good feature from data!

Non-linear predictors

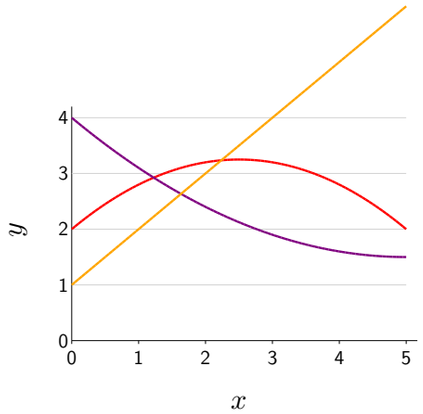
Linear predictors:

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



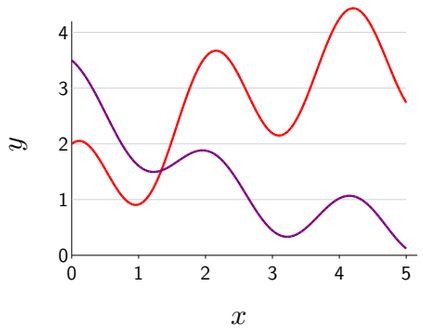
Non-linear (quadratic) predictors:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x), \ \phi(x) = [1, x, x^2]$$



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 neural networks are a way to build complex nonlinear predictors without creating a large number of complex feature extractors by hand
- It is a common misconception that neural networks are somehow more expressive than other models but this isnt necessarily true. You can define ϕ to be extremely large, to the point where it can approximate arbitrary smooth functions – this is the kernel based methods i mentioned last lecture
- 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

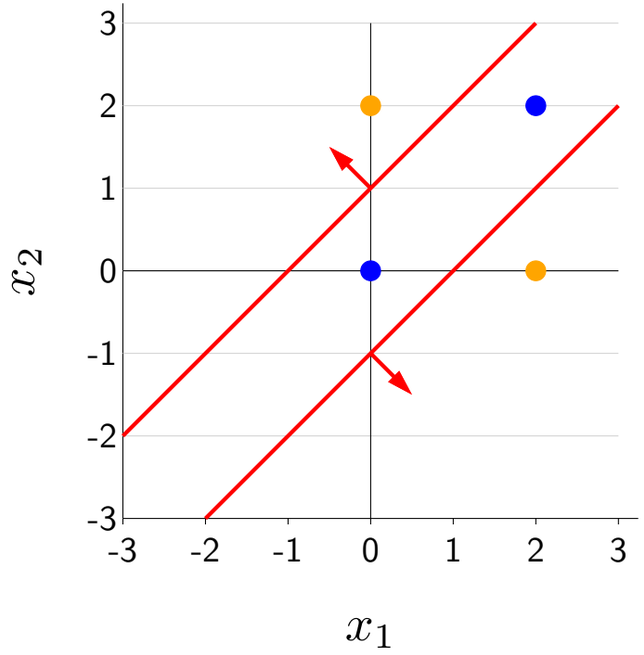
Example: predicting car collision

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:

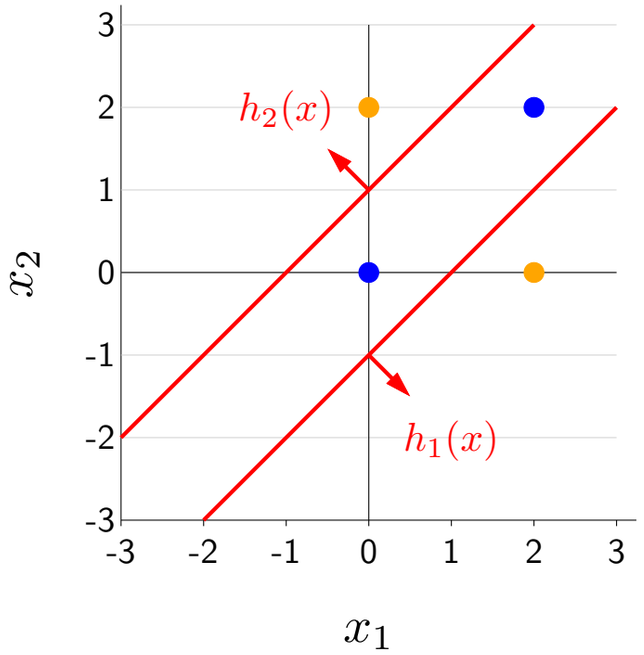
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

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:

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



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!

Learning strategy

Define: $\phi(x) = [1, x_1, x_2]$

Intermediate hidden subproblems:

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

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \qquad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

$$f_{\mathbf{V},\mathbf{w}}(x) = \text{sign}(w_1 h_1 + w_2 h_2) \qquad \mathbf{w} = [1, 1]$$

Key idea: joint learning

Goal: learn both hidden subproblems $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$ and combination weights $\mathbf{w} = [w_1, w_2]$

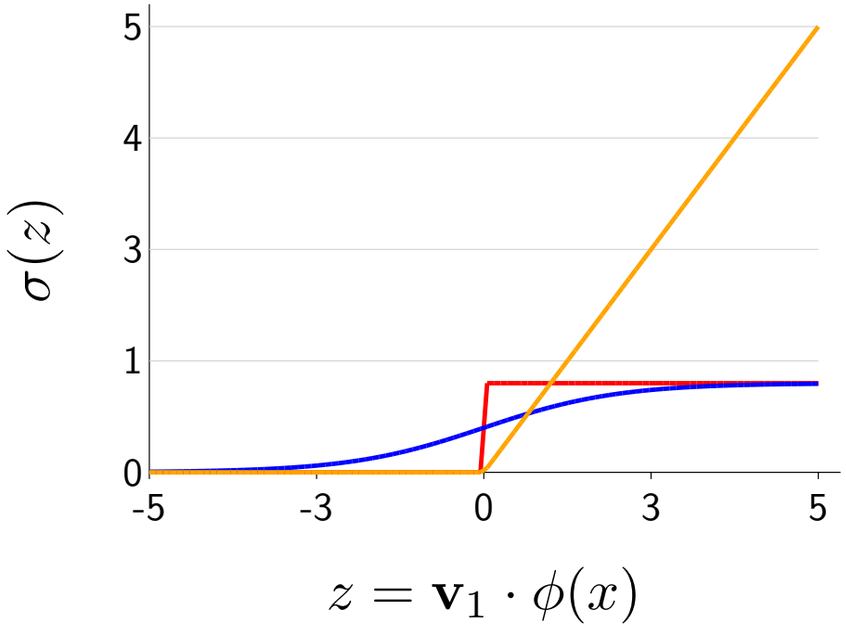
- The story thus far is similar to feature extraction - we have this specific set of h-functions that work
- However, we didnt want to hand craft these things, so what do we do?
- Well, we can just learn these intermediate feature extractors from data

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



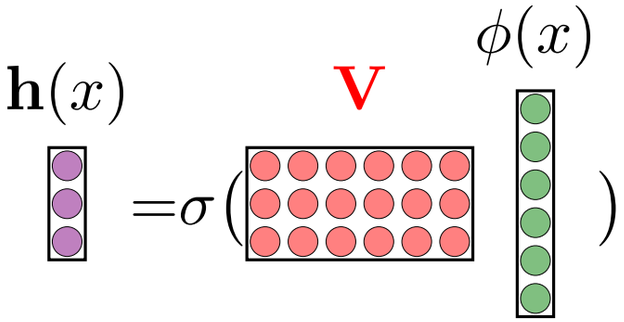
- Threshold: $\mathbf{1}[z \geq 0]$
- Logistic: $\frac{1}{1+e^{-z}}$
- ReLU: $\max(z, 0)$

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

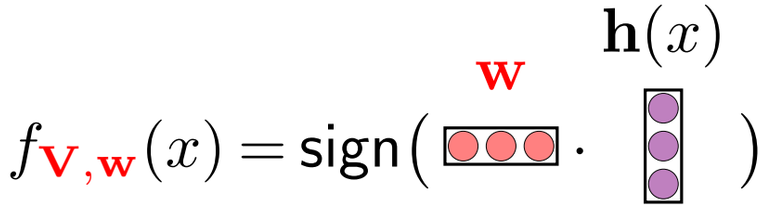
- Later we'll show how to perform learning using gradient descent, but we can anticipate one problem, which we encountered when we tried to optimize the zero-one loss.
- The gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is always zero because of the threshold function.
- To fix this, we replace 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!

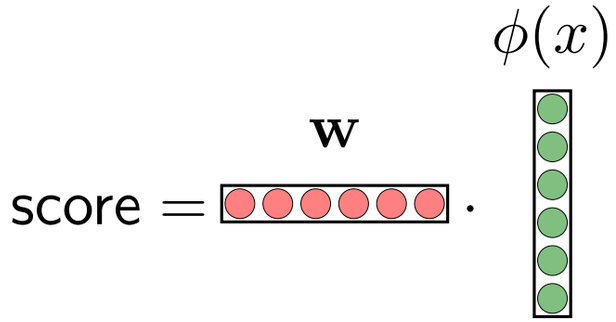
Hypothesis class:

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

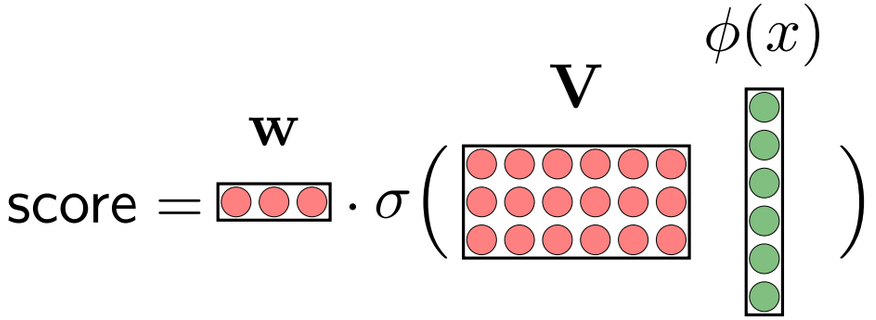
- Now we are finally ready to define the hypothesis class of two-layer neural networks.
- We start with a feature vector $\phi(x)$.
- We multiply it by a weight matrix $\mathbf{V}$ (whose rows can be interpreted as the weight vectors of the k intermediate subproblems.
- Then we apply the activation function 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

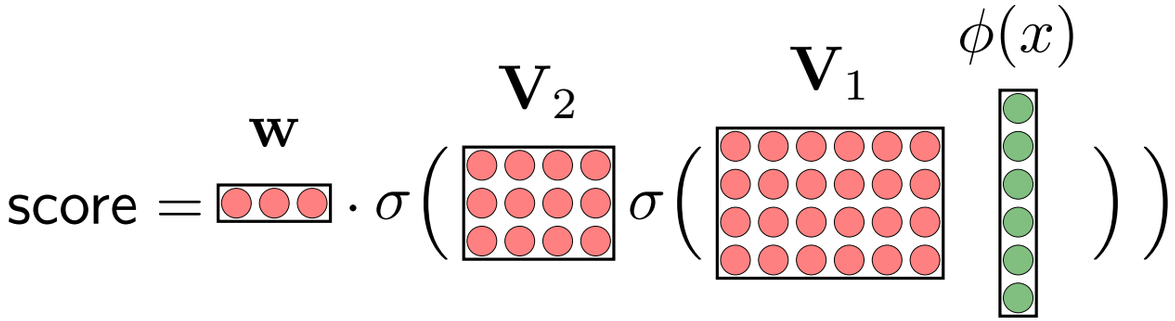
1-layer neural network:



2-layer neural network:



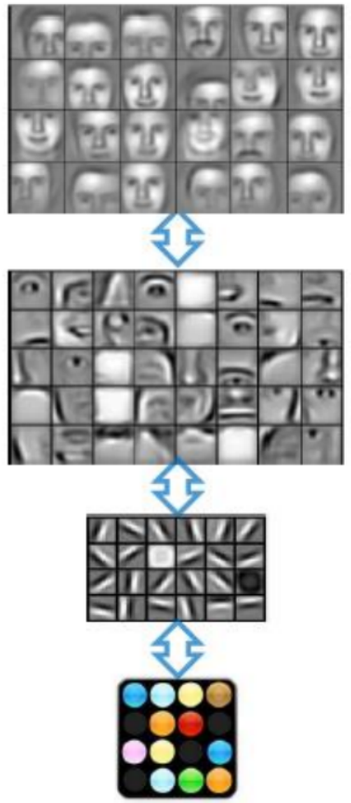
3-layer neural network:



- Why not go further? we currently have a linear classifier on top of a simple feature extractor. We can expand this and put another function in between the two that can combine simple features to make more complex ones. This is the basic idea behind multi-layer neural networks.
- 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



3rd layer
"Objects"

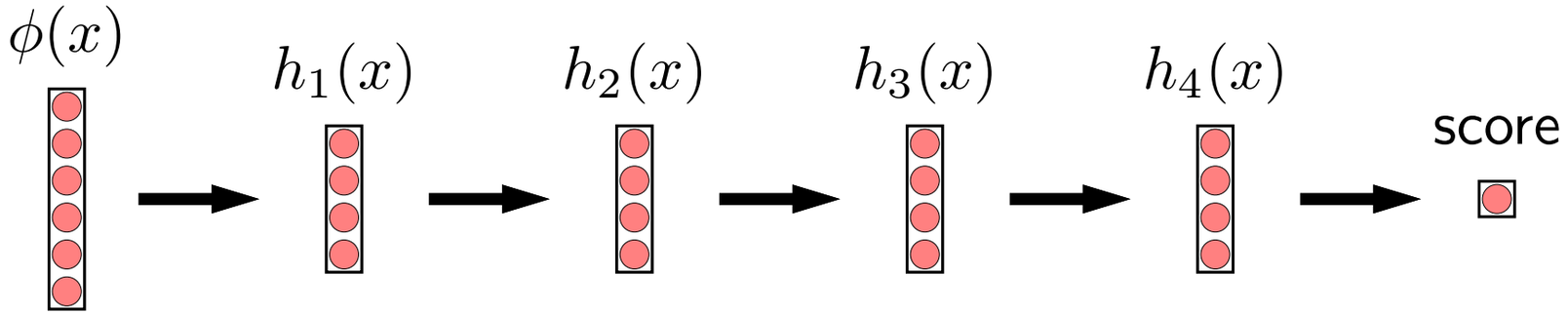
2nd layer
"Object parts"

1st layer
"Edges"

Pixels

- 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 as 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?

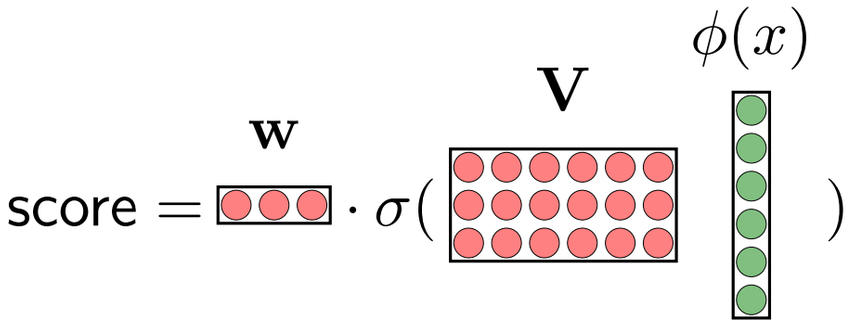


Intuitions:

- 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
- Next up: learning neural networks

- 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.
- Next, we will see how we can learn a neural network by choosing the weights for all the layers.

Roadmap

Topics in the lecture:

Nonlinear features

Feature templates

Neural networks

Backpropagation

- How should we train these models? we are going to use gradient descent
- But it still seems hard to compute the gradient
- We are now going to discuss backpropagation which lets us compute gradients automatically

Motivation: regression with four-layer neural networks

Loss on one example:

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$

Stochastic gradient descent:

$$\mathbf{V}_1 \leftarrow \mathbf{V}_1 - \eta \nabla_{\mathbf{V}_1} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_2 \leftarrow \mathbf{V}_2 - \eta \nabla_{\mathbf{V}_2} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

$$\mathbf{V}_3 \leftarrow \mathbf{V}_3 - \eta \nabla_{\mathbf{V}_3} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

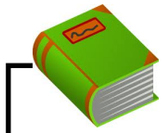
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

How to get the gradient without doing manual work?

- So far, we've defined neural networks, which take an initial feature vector $\phi(x)$ and sends it through a sequence of matrix multiplications and non-linear activations σ. At the end, we take the dot product between a weight vector $\mathbf{w}$ to produce the score.
- In regression, we predict the score, and use the squared loss, which looks at the squared difference betwen the score and the target y.
- Recall that we can use stochastic gradient descent to optimize the training loss (which is an average over the per-example losses). Now, we need to update all the weight matrices, not just a single weight vector. This can be done by taking the gradient with respect to each weight vector/matrix separately, and updating each parameter with the gradient descent update.
- We can now proceed to take the gradient of the loss function with respect to the various weight vector/matrices. You should know how to do this: just apply the chain rule. But grinding through this complex expression by hand can be quite tedious. If only we had a way for this to be done automatically for us...

Computation graphs

$$\text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}_3 \sigma(\mathbf{V}_2 \sigma(\mathbf{V}_1 \phi(x)))) - y)^2$$



Definition: computation graph

A directed acyclic graph whose root node represents the final mathematical expression and each node represents intermediate subexpressions.

Upshot: compute gradients via general **backpropagation** algorithm

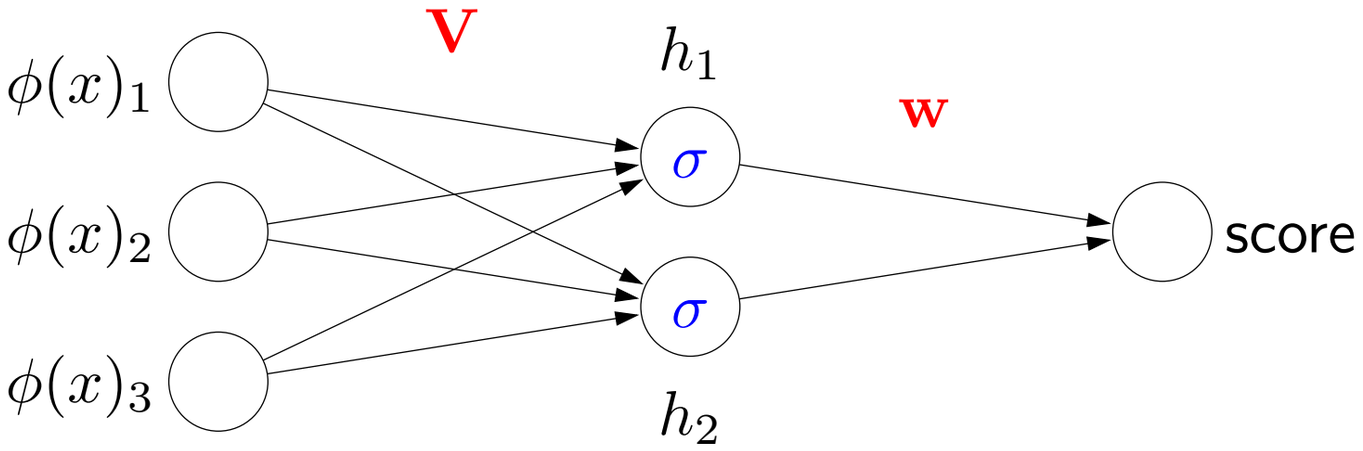
Purposes:

- Automatically compute gradients (how TensorFlow and PyTorch work)
- Gain insight into modular structure of gradient computations

- This is where computation graphs are helpful.
- A computation graph is a directed acyclic graph that represents an arbitrary mathematical expression. The root of that node represents the final expression, and the other nodes represent intermediate subexpressions.
- After having constructed the graph, we can compute all the gradients we want by running the general-purpose backpropagation algorithm, which operates on an arbitrary computation graph.
- There are two purposes to using computation graphs. The first and most obvious one is that it avoids having us to do pages of calculus, and instead delegates this to a computer. This is what packages such as TensorFlow or PyTorch do, and essentially all non-trivial deep learning models are trained like this.
- The second purpose is that by defining the graph, we can gain more insight into the nature of how gradients are computed in a modular way.

Neural networks

Neural network (one hidden layer):

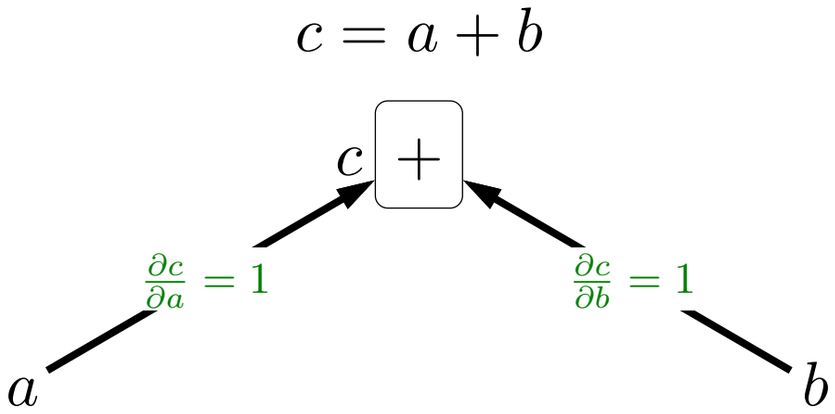


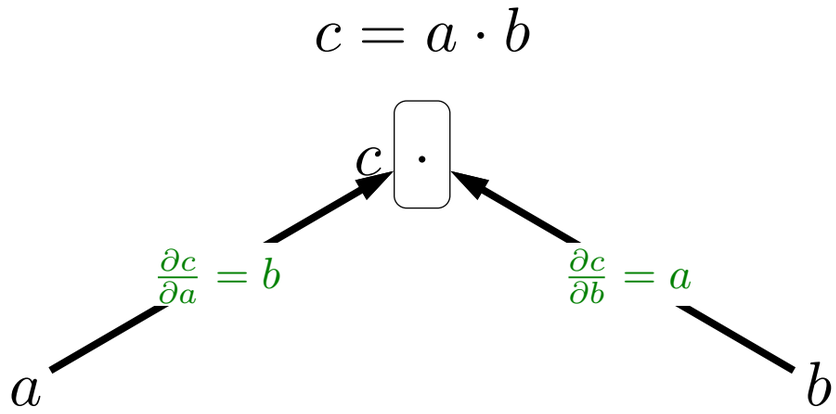
We can represent neural networks as a graph:

- Each node performs a computation
- Each edge is a variable

- Remember that a neural network can be written as a graph
- This is a simple example of a two layer net from earlier
- Backpropagation is a simple set of rules that let us compute gradients by walking backwards from the loss to the paramter we are differentiating
- The key object is a computation graph, which is a slightly more detailed version of this graph you see here

Functions as boxes



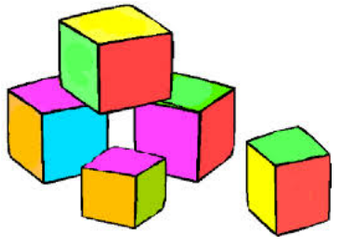


$$(a + \epsilon) + b = c + 1\epsilon$$
$$a + (b + \epsilon) = c + 1\epsilon$$

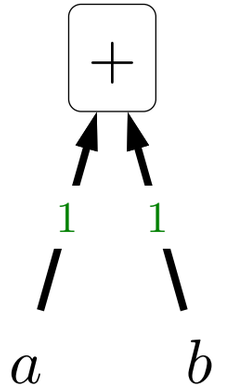
$$(a + \epsilon)b = c + b\epsilon$$
$$a(b + \epsilon) = c + a\epsilon$$

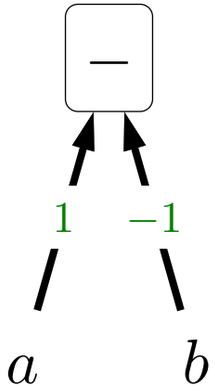
Gradients: how much does c change if a or b changes?

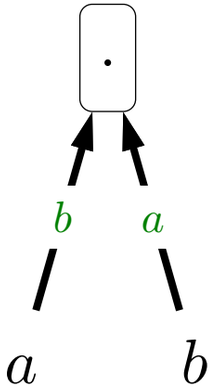
- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output.
- For example, take $c = a + b$. The key question is: if we perturb a by a small amount ϵ, how much does the output c change? In this case, the output c is also perturbed by 1ϵ, so the gradient (partial derivative) is 1. We put this gradient on the edge.
- We can handle $c = a \cdot b$ in a similar way.
- Intuitively, the gradient is a measure of local sensivity: how much input perturbations get amplified when they go through the various functions.

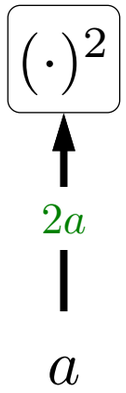


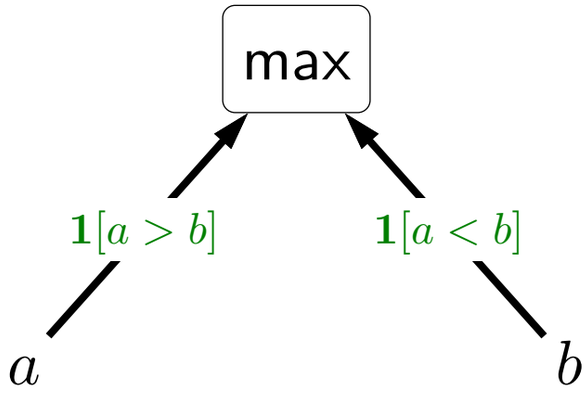
Basic building blocks

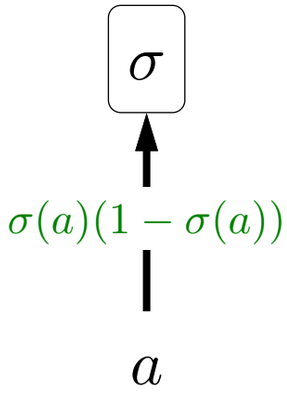






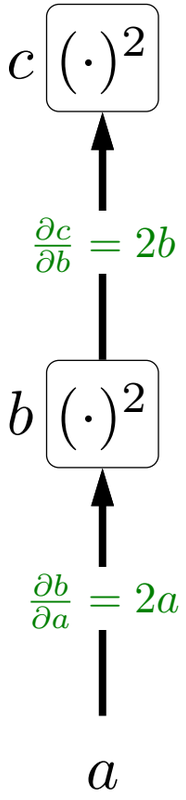






- Here are some more examples of simple functions and their gradients. Let's walk through them together.
- These should be familiar from basic calculus. All we've done is present them in a visually more intuitive way.
- For the max function, changing a only impacts the max iff $a > b$; and analogously for b.
- For the logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$, a bit of algebraic elbow grease produces the gradient. You can check that the gradient is zero when $|a| \to \infty$.
- It turns out that these simple functions are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter.

Function composition

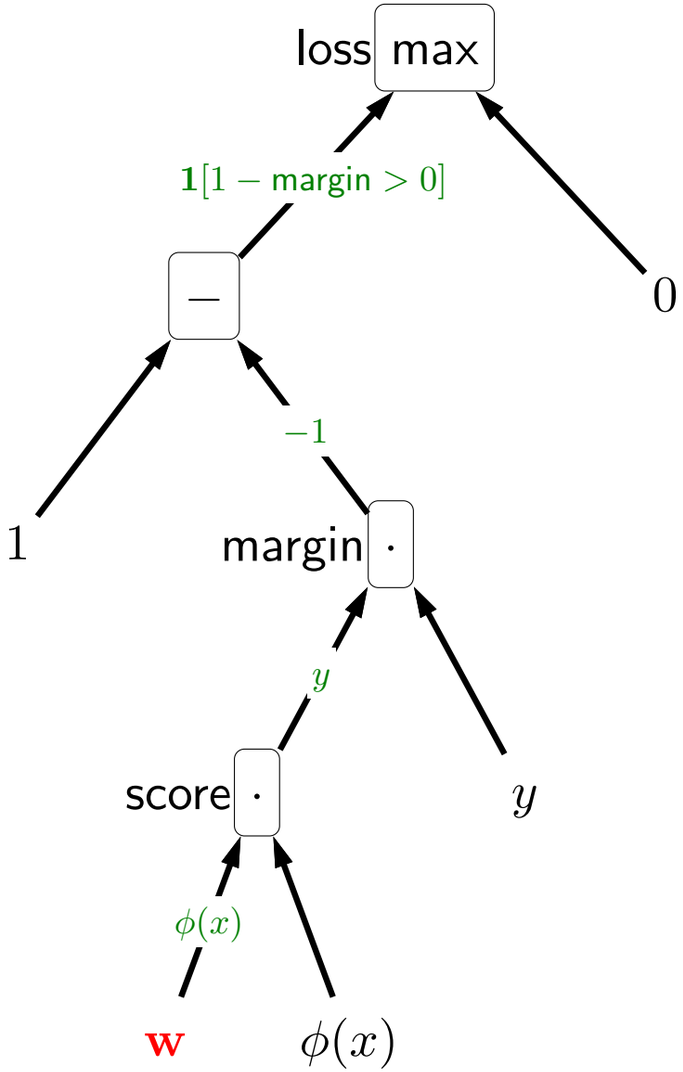


Chain rule:

$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b}\frac{\partial b}{\partial a} = (2b)(2a) = (2a^2)(2a) = 4a^3$$

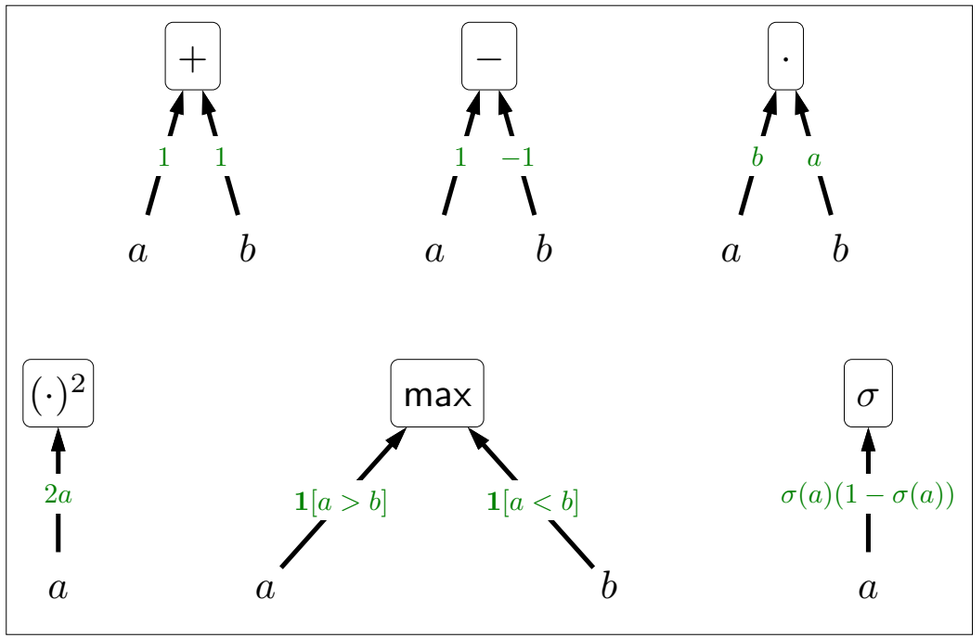
- Given these building blocks, we can now put them together to create more complex functions.
- Consider applying some function (e.g., squared) to a to get b, and then applying some other function (e.g., squared) to get c.
- What is the gradient of c with respect to a?
- We know from our building blocks the gradients on the edges.
- The final answer is given by the **chain rule** from calculus: just multiply the two gradients together.
- You can verify that this yields the correct answer $(2b)(2a) = 4a^3$.
- This visual intuition will help us better understand more complex functions.

Linear classification with hinge loss



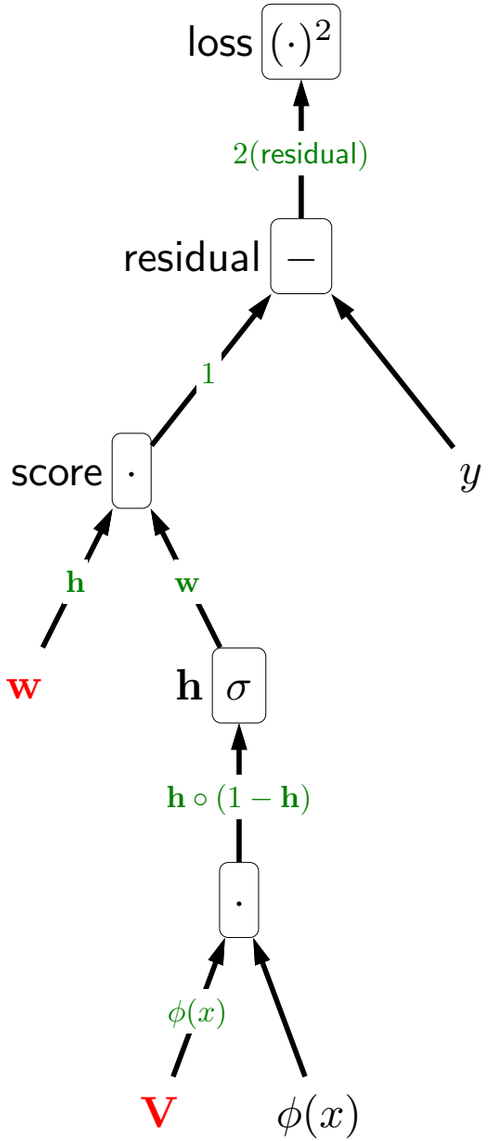
$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = -\mathbf{1}[\text{margin} < 1]\phi(x)y$$



- Now let's turn to our first real-world example: the hinge loss for linear classification. We already computed the gradient before, but let's do it using computation graphs.
- We can construct the computation graph for this expression, proceeding bottom up. At the leaves are the inputs and the constants. Each internal node is labeled with the operation (e.g., $\cdot$) and is labeled with a variable naming that subexpression (e.g., margin).
- In red, we have highlighted the weights $\mathbf{w}$ with respect to which we want to take the gradient. The central question is how small perturbations in $\mathbf{w}$ affect a change in the output (loss).
- We can examine each edge from the path from $\mathbf{w}$ to loss, and compute the gradient using our handy reference of building blocks.
- The actual gradient is the product of the edge-wise gradients from $\mathbf{w}$ to the loss output.

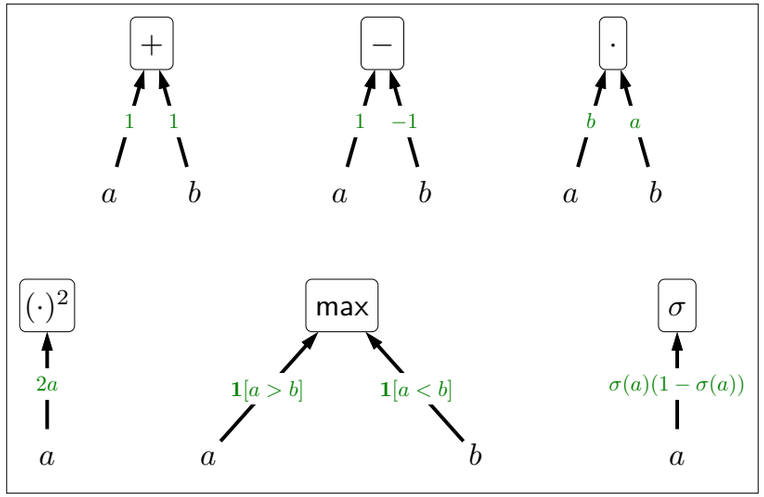
Two-layer neural networks



$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (\mathbf{w} \cdot \sigma(\mathbf{V}\phi(x)) - y)^2$$

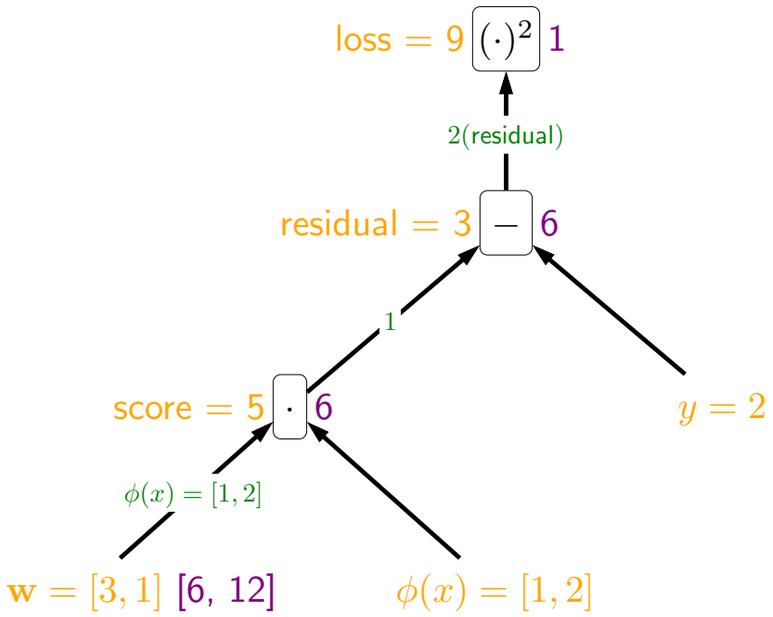
$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

$$\nabla_{\mathbf{V}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{w} \circ \mathbf{h} \circ (1 - \mathbf{h})\phi(x)^\top$$



- We now finally turn to neural networks, but the idea is essentially the same.
- Specifically, consider a two-layer neural network driving the squared loss.
- Let us build the computation graph bottom up.
- Now we need to take the gradient with respect to $\mathbf{w}$ and $\mathbf{V}$. Again, these are just the product of the gradients on the paths from $\mathbf{w}$ or $\mathbf{V}$ to the loss node at the root.
- Note that the two gradients have in common the the first two terms. Common paths result in common subexpressions for the gradient.
- There are some technicalities when dealing with vectors worth mentioning: First, the $\circ$ in $\mathbf{h} \circ (1 - \mathbf{h})$ is elementwise multiplication (not the dot product), since the non-linearity σ is applied elementwise.
- Second, notice the transpose for the gradient expression with respect to $\mathbf{V}$. this looks a bit unusual, but it happens because unlike everything weve done, V is a matrix (not a vector) and we are actually taking jacobians because $\mathbf{V}\phi(x)$ is a vector-valued function.
- You can work through this by hand, but to see that this is right, notice that the gradient of V needs to have the same dimensions as V and this transpose gives this the right dimension.
- This computation graph also highlights the modularity of hypothesis class and loss function. You can pick any hypothesis class (linear predictors or neural networks) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

Backpropagation

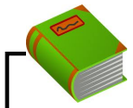


$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$$

backpropagation

$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$$



Definition: Forward/backward values

Forward: f_i is value for subexpression rooted at i

Backward: $g_i = \frac{\partial \text{loss}}{\partial f_i}$ is how f_i influences loss

Algorithm: backpropagation algorithm

Forward pass: compute each f_i (from leaves to root)

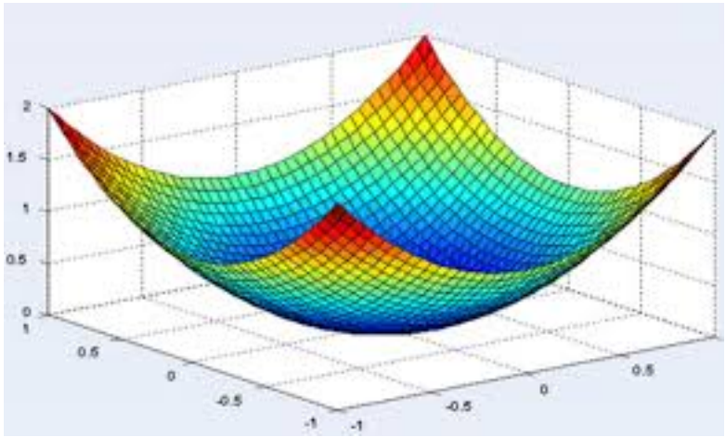
Backward pass: compute each g_i (from root to leaves)

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding.
- Now let us introduce the **backpropagation** algorithm, a general procedure for computing gradients given only the specification of the function.
- Let us go back to the simplest example: linear regression with the squared loss.
- All the quantities that we've been computing have been so far symbolic, but the actual algorithm works on real numbers and vectors. So let's use concrete values to illustrate the backpropagation algorithm.
- The backpropagation algorithm has two phases: forward and backward. In the forward phase, we compute a **forward value** f_i for each node, coresponding to the evaluation of that subexpression.
- Let's work through the example.
- In the backward phase, we compute a **backward value** g_i for each node. This value is the gradient of the loss with respect to that node, which is also the product of all the gradients on the edges from the node to the root. To compute this backward value, we simply take the parent's backward value and multiply by the gradient on the edge to the parent. Let's work through the example.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but have the same dimensionality.

A note on optimization

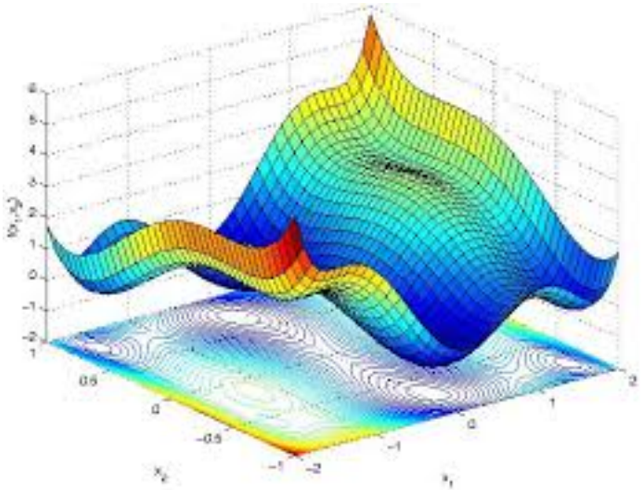
$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

Neural networks

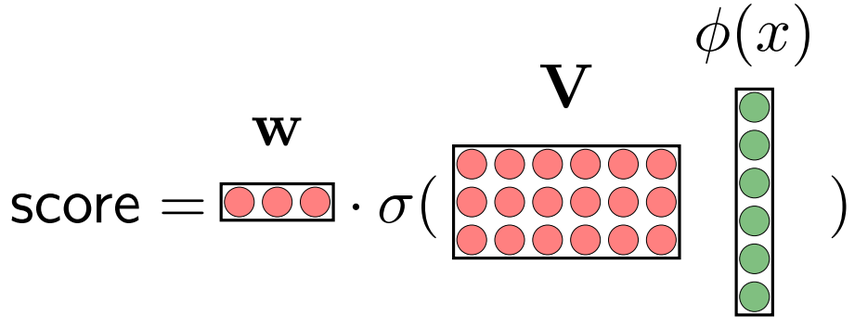


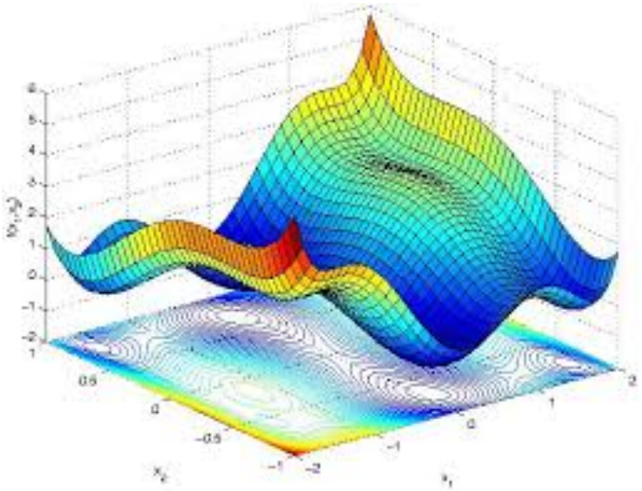
(non-convex)

Optimization of neural networks is in principle hard

- So now we can apply the backpropagation algorithm and compute gradients, stick them into stochastic gradient descent, and get some answer out.
- One question which we haven't addressed is whether stochastic gradient descent will work in the sense of actually finding the weights that minimize the training loss.
- For linear predictors (using the squared loss or hinge loss), TrainLoss$(\mathbf{w})$ is a convex function, which means that SGD (with an appropriate step size) is theoretically guaranteed to converge to the global optimum.
- However, for neural networks, TrainLoss$(\mathbf{V}, \mathbf{w})$ is typically non-convex which means that there are multiple local optima, and SGD is not guaranteed to converge to the global optimum. There are many settings that SGD fails both theoretically and empirically, but in practice, SGD on neural networks can work much better than theory would predict, provided certain precautions are taken. The gap between theory and practice is not well understood and an active area of research.

How to train neural networks



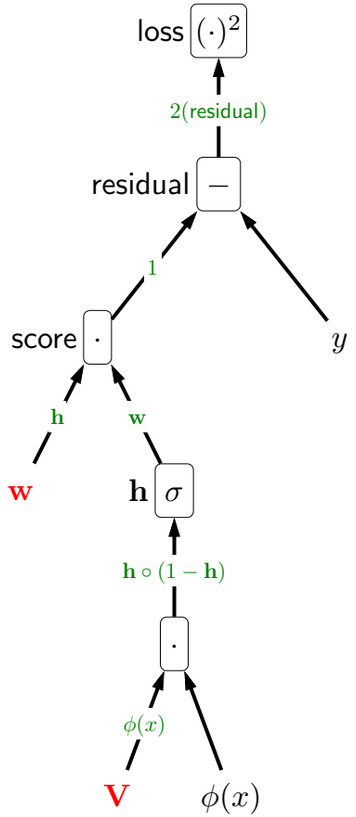


- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don't let gradients vanish or explode!

- Training a neural network is very much like driving stick. In practice, there are some "tricks" that are needed to make things work properly. Just to name a few to give you a sense of the considerations:
- Initialization (where you start the weights) matters for non-convex optimization. Unlike for linear models, you can't start at zero or else all the subproblems will be the same (all rows of $\mathbf{V}$ will be the same). Instead, you want to initialize with a small amount of random noise.
- It is common to use overparameterized neural networks, ones with more hidden units (k) than is needed, because then there are more "chances" that some of them will pick out on the right signal, and it is okay if some of the hidden units become "dead".
- There are small but important extensions of stochastic gradient descent that allow the step size to be tuned per weight.
- Perhaps one high-level piece of advice is that when training a neural network, it is important to monitor the gradients. If they vanish (get too small), then training won't make progress. If they explode (get too big), then training will be unstable.

Summary



- Computation graphs: visualize and understand gradients
- Backpropagation: general-purpose algorithm for computing gradients

- The most important concept in this part of the lecture is the idea of a **computation graph**, which allows us to represent arbitrary mathematical expressions, by utilizing simple building blocks. They hopefully have given you a more visual and better understanding of what gradients are about.
- The **backpropagation** algorithm allows us to simply write down an expression, and never have to take a gradient manually again. However, it is still important to understand how the gradient arises, so that when you try to train a deep neural network and your gradients vanish, you know how to think about debugging your network.
- The generality of computation graphs and backpropagation makes it possible to iterate very quickly on new types of models and loss functions