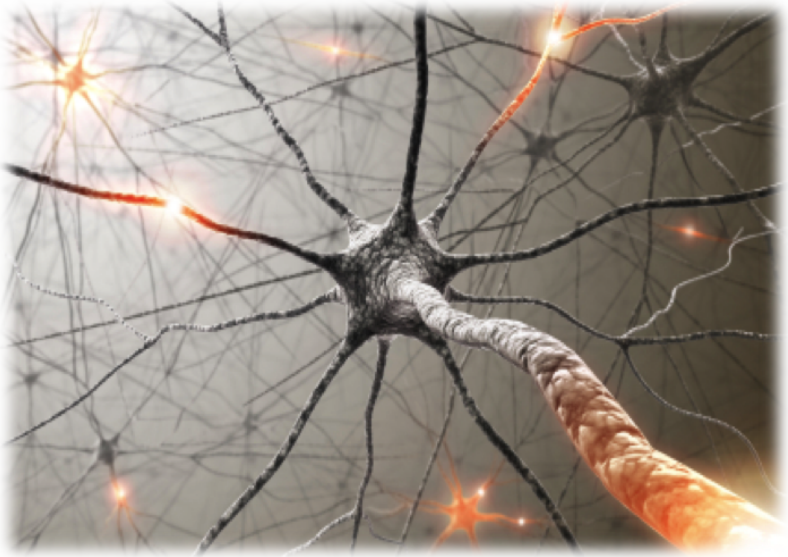
Machine Learning II: Non-Linear Models



Roadmap

Topics in the lecture:

Nonlinear features

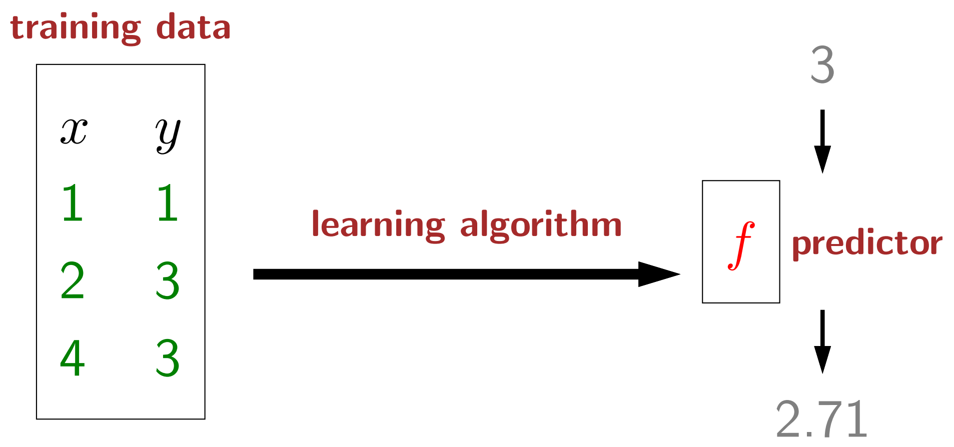
Feature templates

Neural networks

Backpropagation

- In this lecture, we will cover non-linear predictors. We will start with 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
- Then, we will cover the basics of neural networks, defining neural networks as a model and discussing how to compute gradients for these models using backpropagation.
- As a reminder, feel free to raise your hand or post in the Ed thread when you have a question, and we will discuss the questions throughout the lecture. If you would like to ask a question anonymously, feel free to send a private message to Mike. We have about 55 minutes of material today, which leaves us with about 25 minutes for questions throughout!
- Lets get started with nonlinear features.

Linear regression



training data

x	y
1	1
2	3
4	3

Which predictors are possible?

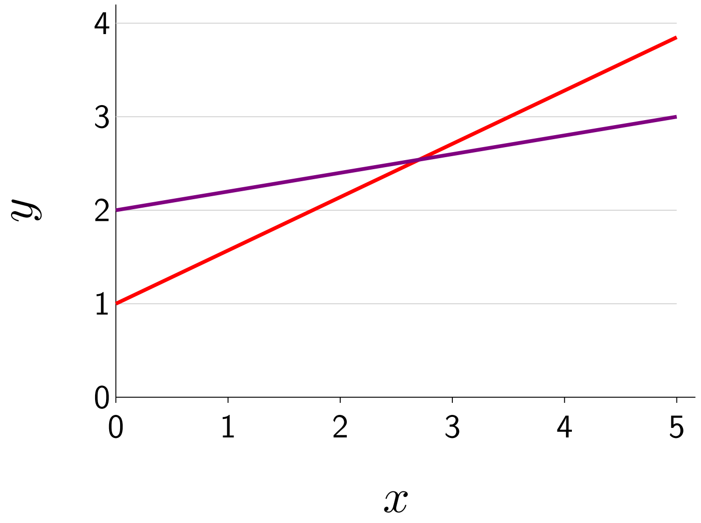
Hypothesis class

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

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

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

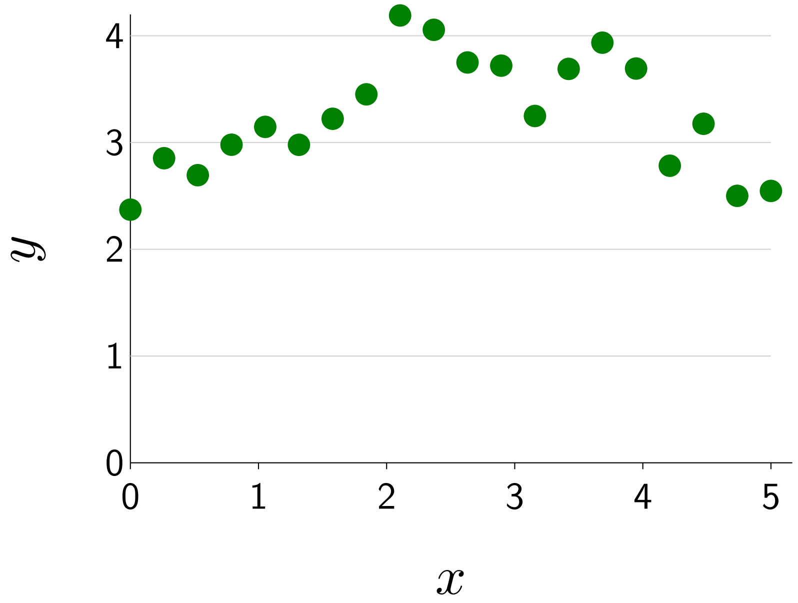
$f(x) = [2, 0.2] \cdot \phi(x)$



- We will look at regression and later turn to classification.

- 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 or non-parametrics.
- But let's see how far we can get with the machinery of linear predictors first, and you might be surprised with how far we can go.

Quadratic predictors

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

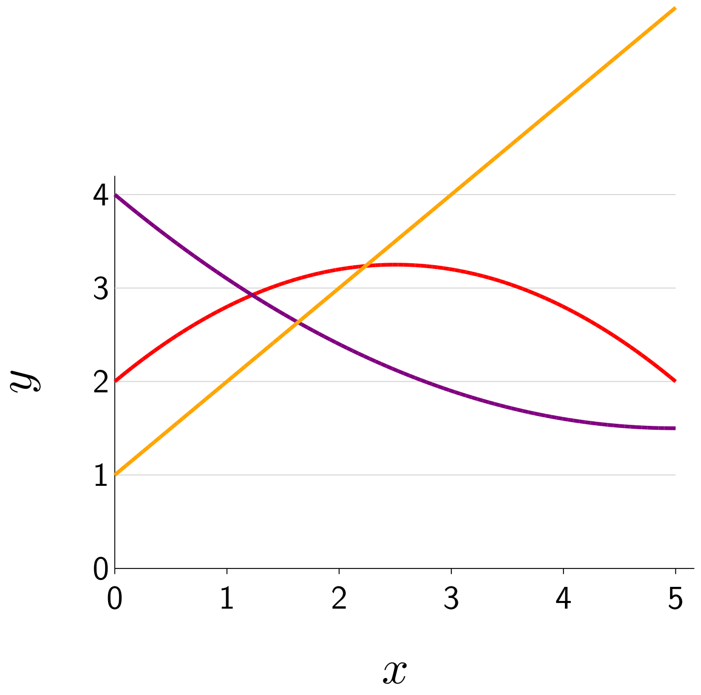
Example: $\phi(3) = [1, 3, 9]$

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

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

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

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



Non-linear predictors just by changing ϕ

- A linear model takes some input and makes a prediction that is linear in its inputs, by taking a dot product. But there is no rule saying that we cannot non-linearly transform the input data first, which is where the function ϕ comes in.
- 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 our features ϕ with 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, such as with the 1,1,0 feature vector, 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. So our model is strictly more expressive, though potentially at the cost of more samples.
- In summary, we've now seen our first example of obtaining non-linear predictors just by changing the feature extractor ϕ!
- As an additional note, I would encourage you to try thinking about how to use this idea to learn two- or even three- dimensional quadratics. What new challenges arise? It is going to turn out you will need not just squared terms for each dimension but also all of the cross terms.

Piecewise constant predictors

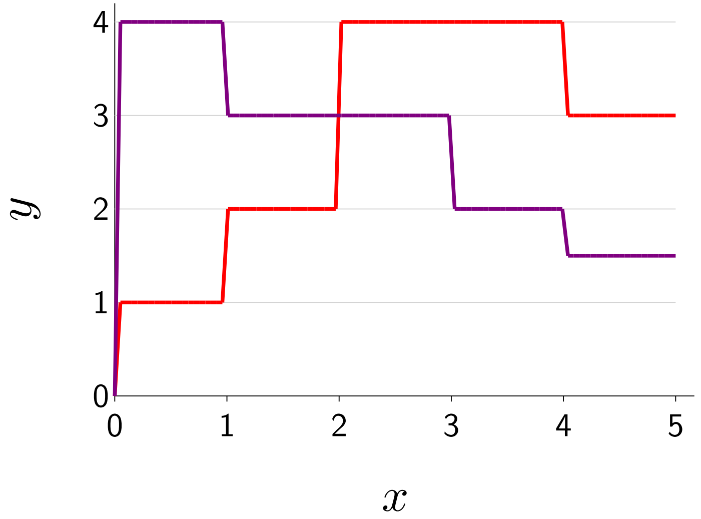
$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]$$

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 nice but still a bit restricted: they can only go up and then down smoothly (or vice-vera).
- We can introduce another type of feature extractor that divides the input space into regions and allows the predicted value of each region to vary independently, This allows you to represent 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, allowing you to model richer and richer functions. In the limit, you can essentially capture any predictor you want.
- This seems quite nice, but what is the catch? The drawback of these kinds of features is that you need alot of training data to learn these predictors. You need a sample within each interval to know its value, and there is no sharing of information across these intervals. Doing this trick in d-dimensions rather than 1-dimension, your data requirements will go up exponentially in the dimension d!

Predictors with periodicity structure

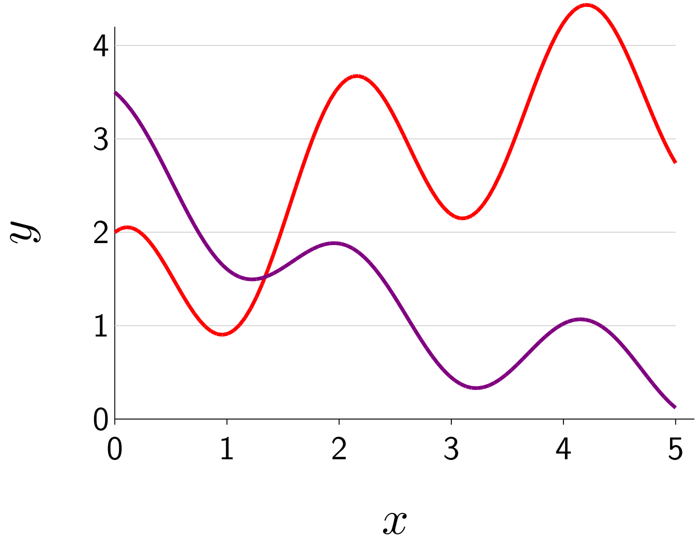
$$\phi(x) = [1, x, x^2, \cos(3x)]$$

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

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

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

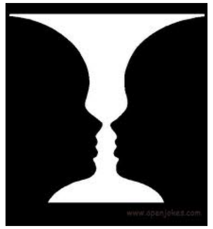
$\mathcal{F} = \{f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) : \mathbf{w} \in \mathbb{R}^4\}$



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 or sine to capture these trends.
- In this case, we can simply append the cosine of $3x$ to feature vector to represent such periodic functions at a particular frequency.
- 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 and let the learning algorithm decide what to use.
- Features represent what properties **might** be useful for prediction. If a feature is not useful, then the learning algorithm can (and hopefully will) assign a weight close to zero to that feature. Of course, the more features one has, the harder learning becomes, and the more samples you will need.

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

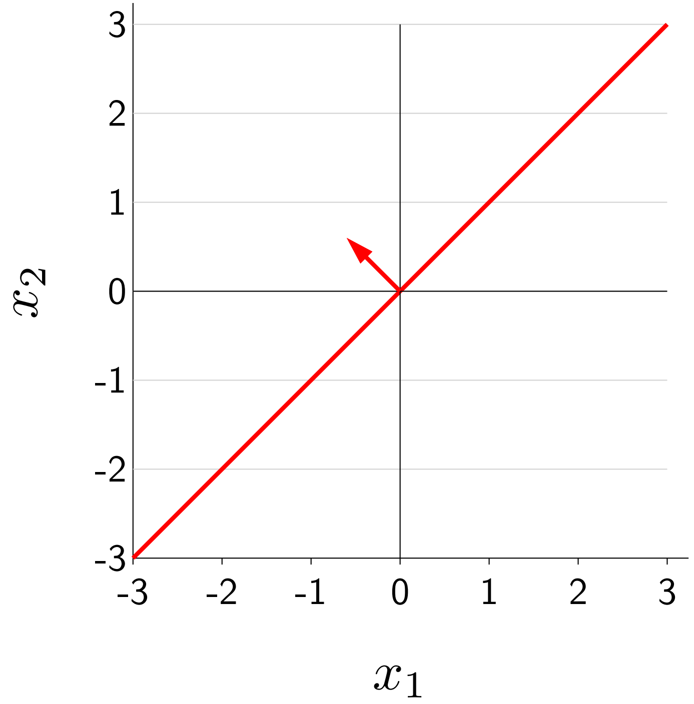
- Expressivity: score $\mathbf{w} \cdot \phi(x)$ can be a **non-linear** function of x
- Efficiency: score $\mathbf{w} \cdot \phi(x)$ always a **linear** function of $\mathbf{w}$

- Stepping back, how are we able to obtain non-linear predictors if we're still using the machinery of linear predictors? This is perhaps a bit surprising! You can think of this like pre-processing: the feature map ϕ is a way of transforming the inputs so that the function looks linear in this new space.
- Importantly, the score $\mathbf{w} \cdot \phi(x)$ is linear in the weights $\mathbf{w}$ and features $\phi(x)$. However, the score is not linear in x (it might not even make sense because x need not be a vector of numbers 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 and sample complexity. We have to learn more weights, one weight for each feature, which will require larger datasets
- The machinery of non-linear features combines the benefits of linear models, like simplicity and ease of optimization, with the benefits of more expressive 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

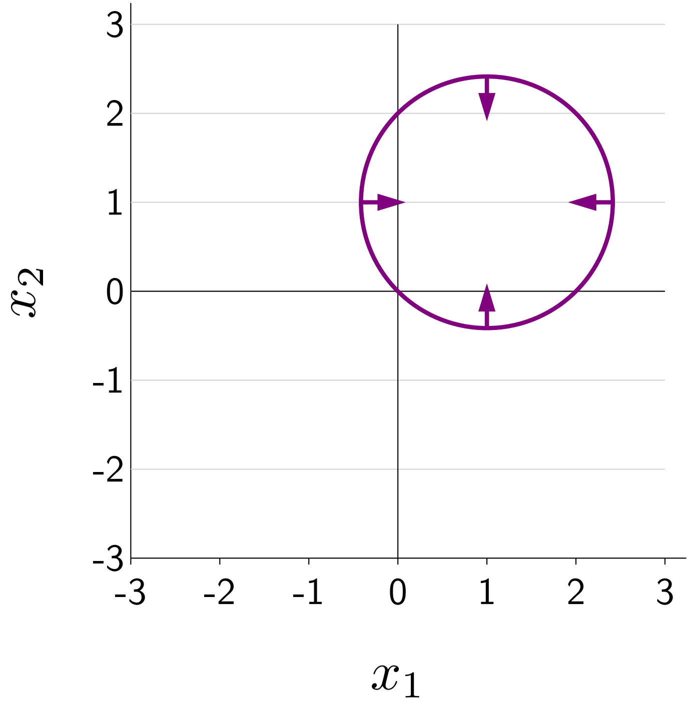
- Now let's turn from regression to classification.
- The story is pretty much the same: you can define arbitrary features to yield non-linear classifiers.
- Recall that in binary classification, the classifier (predictor) returns the sign of the score.
- The score is linear in the weights and the features, but the predictor is already not linear because of the sign operation. So, what do we mean by a classifier being non-linear or linear?
- Since it's not helpful to talk about the linearity of f, we instead ask – can we make the classifier's decision boundary non-linear?
- Last week, we only saw examples of classifiers with decision boundaries that are a straight line. Our goal now is to find classifiers that separate the space of positives and negatives not by a line, but with a more complex boundary.

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:

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



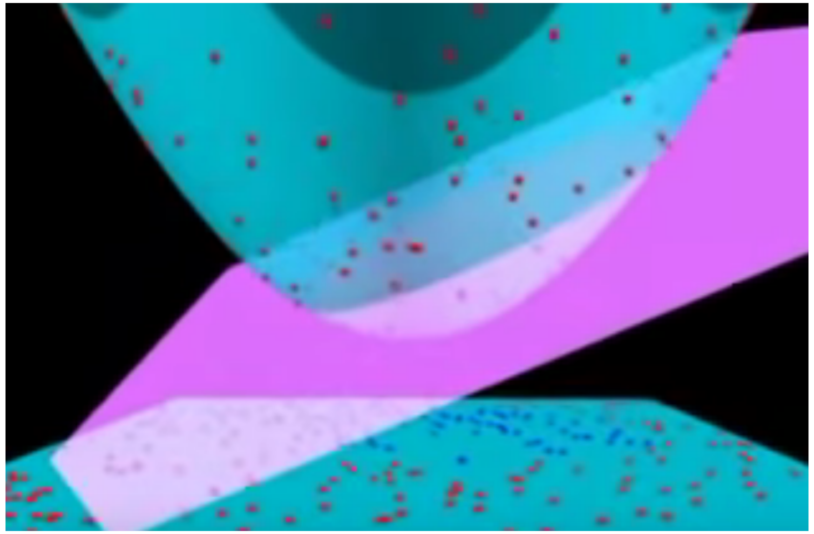
Decision boundary is a circle

- Let us see how we can define a classifier with a non-linear decision boundary but still using the machinery of linear classifiers.
- Let's try to construct a feature extractor that induces a decision boundary that is a circle: where everything inside is classified as $+1$ and everything outside is classified as -1.
- As we saw before, we can add new features to get non-linear functions. In this case, we will add a new feature $x_1^2 + x_2^2$ into the feature vector, and claim that a linear classifier with these features will give us a predictor with a circular decision boundary.
- If you group these terms, then you can 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.
- To recap, all we needed to do to get this decision boundary was to add these quadratic features into our linear classifier.

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



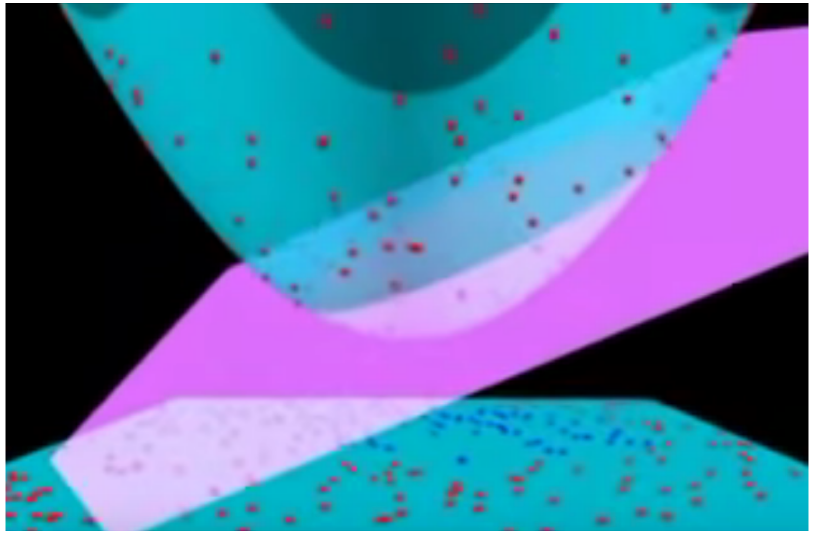
- Let's try to understand the relationship between the non-linearity in x and linearity in $\phi(x)$.
- Click on the image to see the linked video (which is about polynomial kernels and SVMs, but the same principle applies here).
- In the input space x, the decision boundary which separates the red and blue points is a circle.
- We can also visualize the points in **feature space**, where each point is given an additional dimension $x_1^2 + x_2^2$.
- When we add a feature, we are essentially lifting the datapoints into a 3-dimensional space.
- 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, despite being a linear function in the lifted three 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" can be ambiguous: a predictor in regression can be non-linear in the input x but is so far always linear with respect to 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, periodic predictors).
- The takeaway is that linear models are surprisingly powerful! More sophisticated versions of this which are called kernel methods drove much of the statistical machine learning gains in the early 2000s!

Roadmap

Topics in the lecture:

Nonlinear features

Feature templates

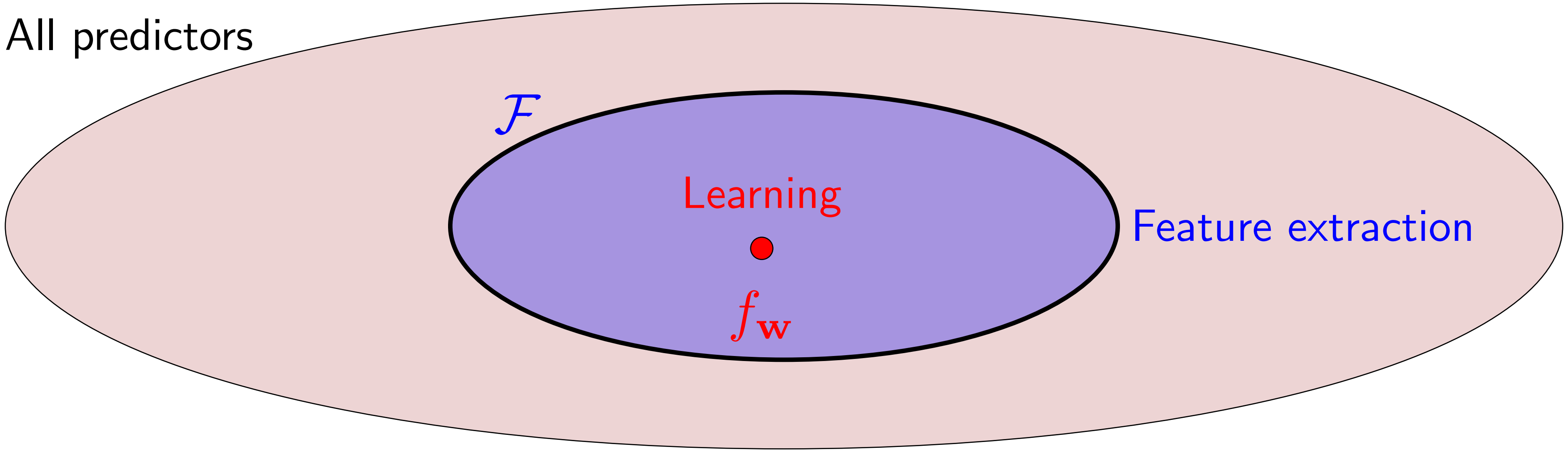
Neural networks

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 one way of defining flexible families of features

Feature extraction + learning

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



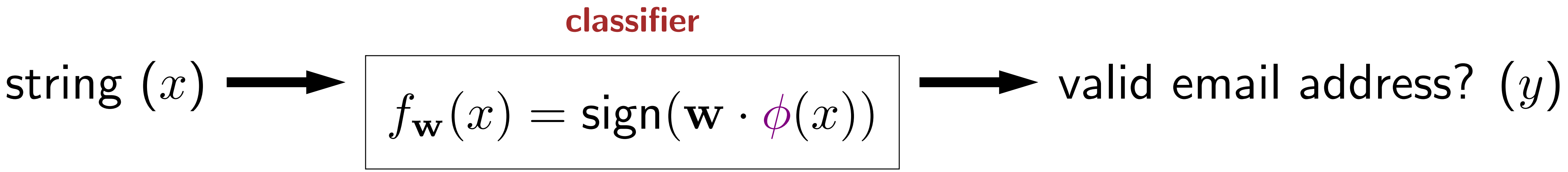
- Feature extraction: choose $\mathcal{F}$ based on domain knowledge
- Learning: choose $f_{\mathbf{w}} \in \mathcal{F}$ based on data

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, the hypothesis class $\mathcal{F}$ contains 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, shown in blue in this figure.
- Learning is the process of choosing a particular predictor $f_{\mathbf{w}}$ from $\mathcal{F}$ given training data.
- But the question that we are currently concerned about is how do we choose $\mathcal{F}$? We saw some options already: the set of linear predictors, the set of quadratic predictors which is a strict superset, 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 predictors as well, since the learning algorithm can avoid them. But we also don't want $\mathcal{F}$ to be too big, or else learning becomes hard, not just computationally but also statistically (which we'll talk about more in the module on generalization).
- In essence, we want the smallest hypothesis class size that still contains good predictors.

Feature extraction with feature names

Example task:

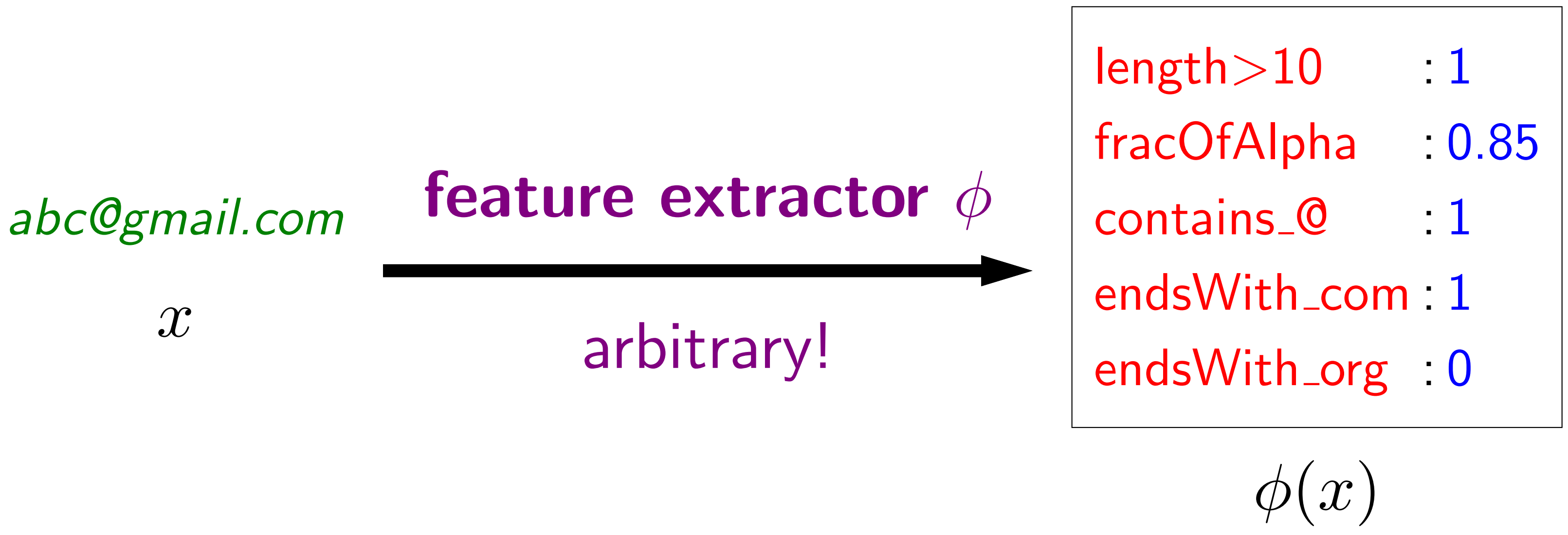


classifier

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 (as 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, but it is helpful for interpreting them within our overall system.

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

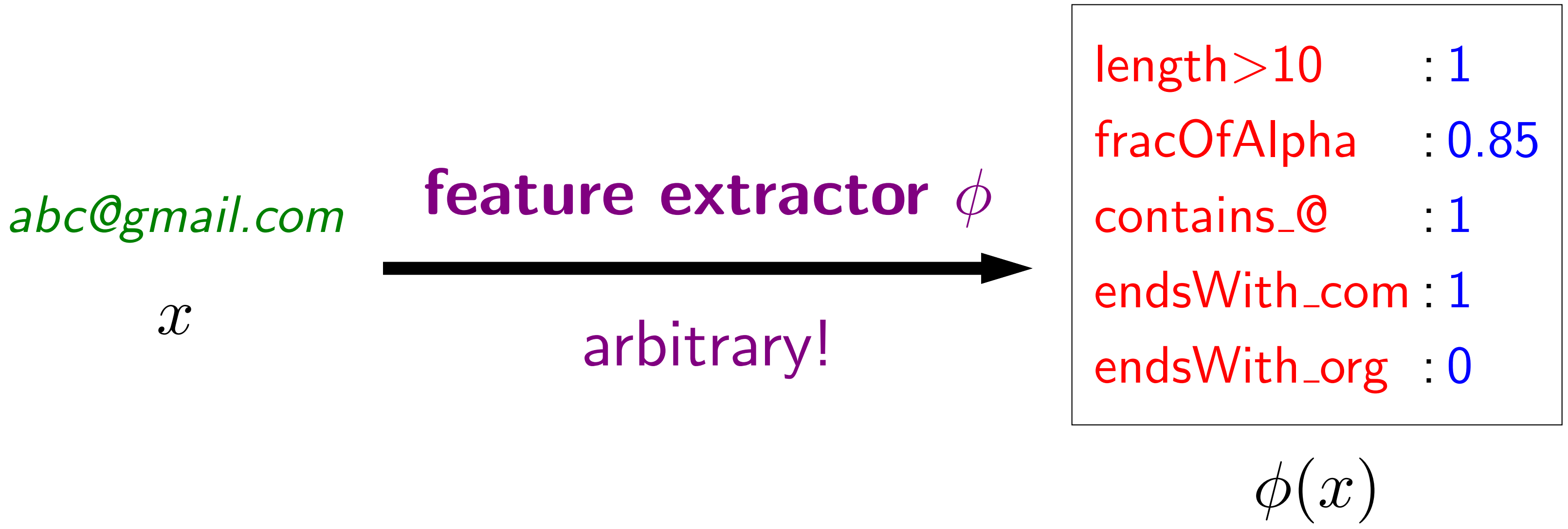
Score: weighted combination of features

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

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 give each weight the corresponding name as well.
- Since the score is simply the dot product between the weight vector and the feature vector. each feature weight w_j determines how the corresponding feature value $\phi_j(x)$ contributions to the prediction, and these contributions are aggregated in the score.
- 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.
- Finally, keep in mind that these contributions are considered holistically. It might be tempting to think that a positive weight means that a feature is always positively correlated with the output. That is not necessarily true - 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 - which you can look into if you find this intriguing.

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

$\phi(x)$

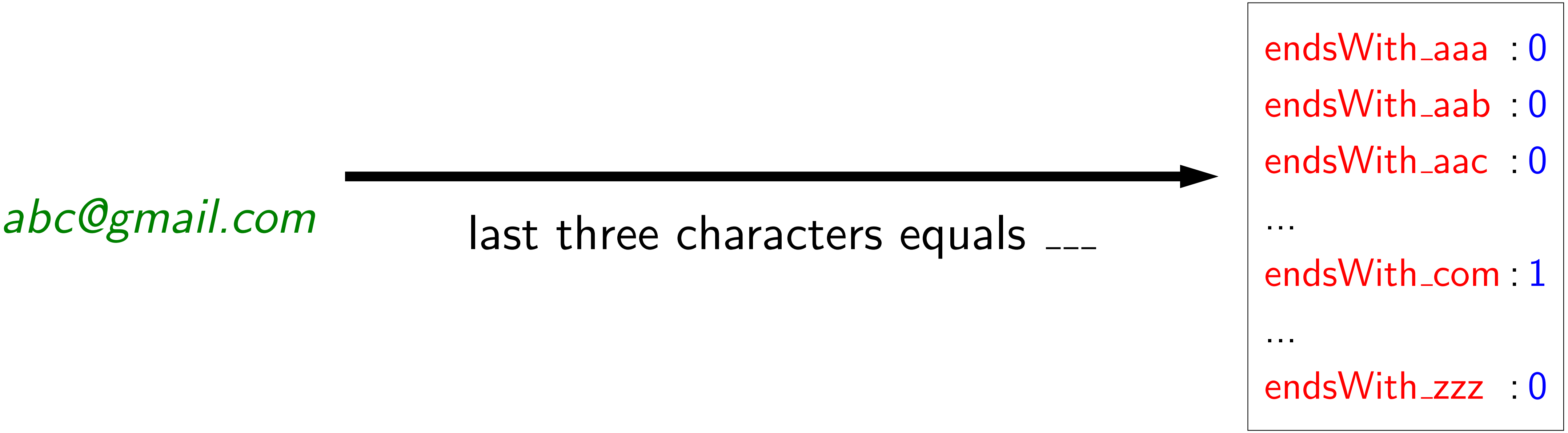
Which features to include? Need an organizational principle...

- How would we go about about creating good features?
- Here, we used our prior knowledge to define certain features (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 (e.g. country codes in the ends with), and there might be other features which are predictive but not intuitive.
- So, we need a more systematic way to go about constructing these features to try to capture all the features we might want.

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 actually mean ”feature template”.)
- Rather than defining individual features like endsWith_com, we can define a single feature template that 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. As long as this set can capture useful templates like .com, then this feature template will be useful
- It is then up to the learning algorithm to figure out which patterns are useful for predicting email addresses by assigning the appropriate feature weights.

Feature templates example 1

Input:

abc@gmail.com

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

- 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) or within some discrete set (e.g. 1, ..., 10), 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 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 box over latitude and longitude. This is very similar to the piecewise constant predictors that we discussed earlier in the lecture, where we have indicators for points that lie in a particular region.

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**.
- For example, in the email address prediction problem, we see that the vast number of endswith features are zeros for this address. In fact, only one of them can be one for this feature template.
- Of course, different feature vectors will have different non-zero features.
- But, in these cases, it is very inefficient to represent all the features explicitly, especially when you are using many different sparse feature templates. Instead, we can store only the values of the non-zero features, assuming all other feature values are zero by default. This compact representation is shown at the bottom.

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
...	

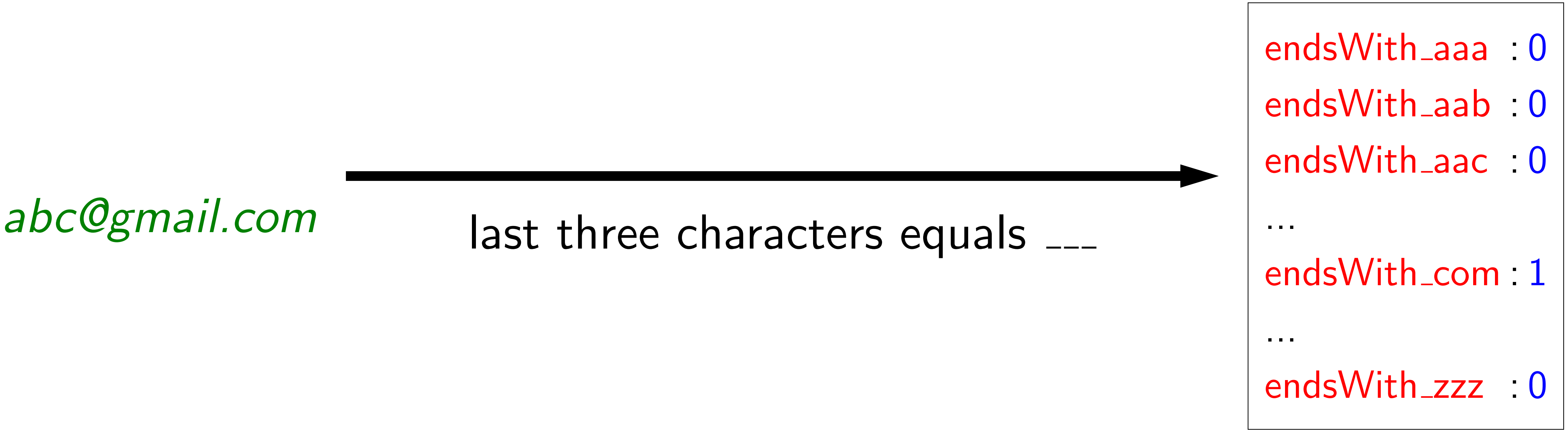
```
{"fracOfAlpha": 0.85, "contains_@": 1}
```

- In general, there are two common ways to implement feature vectors: using arrays (which are good for dense features) and using dictionaries (which are good for sparse features).
- **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). Pixel intensity is a good example, since there is a wide range of intensities and most locations have a non-zero pixel intensity.
- However, when we have sparsity, it is typically much more efficient to implement the feature vector as a **dictionary** (or a 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 areas like natural language processing with linear predictors, and is what allows us to work efficiently over millions of features and words because most of them never appear in an individual example. 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

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

Feature template:



abc@gmail.com

last three characters equals ___

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 saw earlier that the feature extractor can dramatically change the kinds of functions that we can reprensent.
- 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 usually quite domain-specific.
- More powerful predictors such as neural networks will alleviate a lot of the burden of feature engineering (which is one of the reasons it is so popular), but even neural networks use feature vectors as the initial starting point, and therefore their effectiveness is still affected by the feature representation.

Roadmap

Topics in the lecture:

Nonlinear features

Feature templates

Neural networks

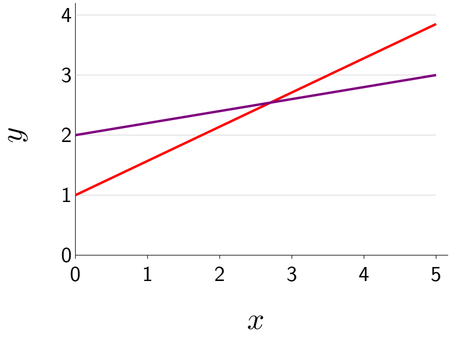
Backpropagation

- Now let's dive into neural networks, first by understanding them as a model and hypothesis class.

Non-linear predictors

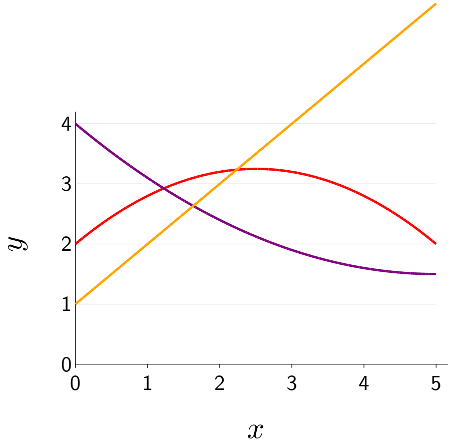
Linear predictors:

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



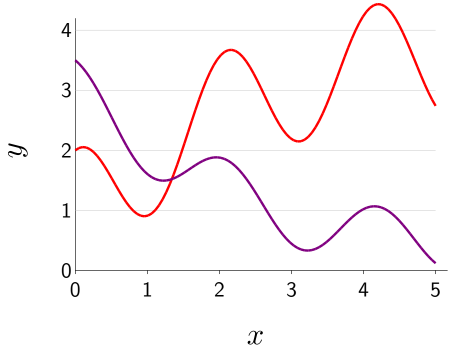
Non-linear (quadratic) predictors:

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



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 predictors that are non-linear in x by simply changing the feature extractor ϕ. For example, adding the feature x^2 gave us quadratic predictors.
- One disadvantage of this approach is that if x were d-dimensional, one would need $O(d^2)$ features and corresponding weights to learn an arbitrary quadratic in d dimensions. This can present 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
- What do neural networks get us? A common misconception that neural networks are 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 idea of using feature maps to approximate complex functions is called a kernel based method. So, there is no inherent benefit in terms of expressive power to using neural networks.
- Rather, one core benefit of neural networks is that they yield non-linear predictors in a more **compact** way and an often more computationally-tractable way. For instance, you might not need $O(d^2)$ features to represent the desired non-linear predictor.

Motivating example

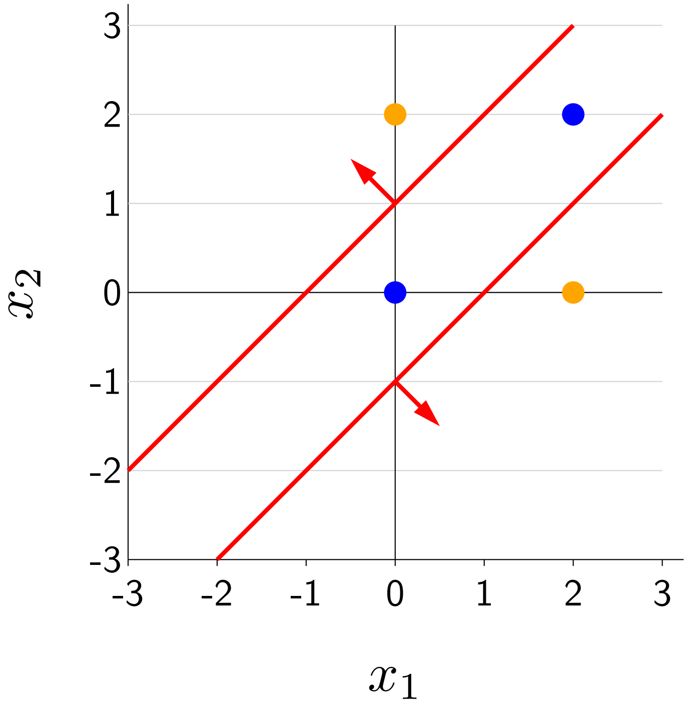
Example: predicting car collision

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. So, we want to learn the decision boundary that separates the blue points from the yellow points. (This is essentially the famous XOR problem that was impossible to fit using linear classifiers. And you can see that no single line will separate these points.)

Decomposing the problem

Test if car 1 is far right of car 2:

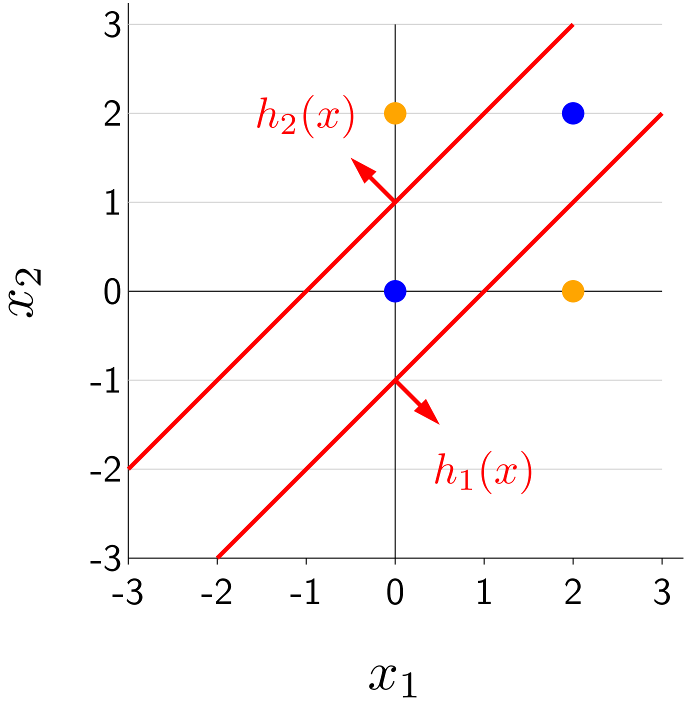
$$h_1(x) = \mathbf{1}[x_1 - x_2 \geq 1]$$

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 is the idea of **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, which is also linear.
- 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 $h_2(x)$, the subproblems, and then simply adding the two outputs. This is one way to compose the results of each subproblem into $f(x)$.
- Note that everything here is basically linear, but we have stacked these components to form a non-linear predictor.

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:

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

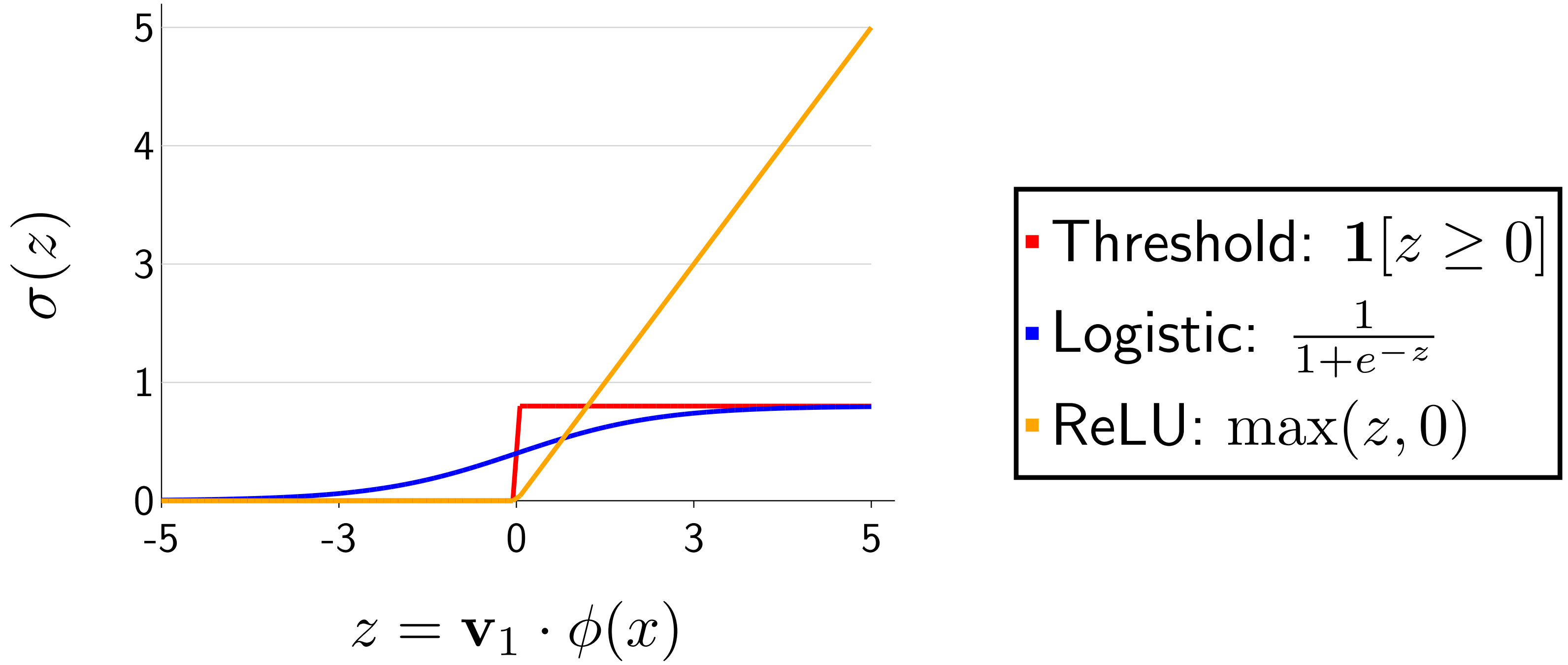
- Now let us rewrite this predictor $f(x)$ using vector notation. This will be helpful as we go forward and define neural networks more formally.
- We can define a feature vector $[1, x_1, x_2]$ and a corresponding weight vector, where the dot product thresholded yields exactly $h_1(x)$.
- We do the same for $h_2(x)$.
- We put the two subproblems into one single equation by stacking the weight vectors into one matrix, to produce a vector-valued function $\mathbf{h}$ that combines h_1 and h_2. We can do this since the input feature vector is the same. You can see the equivalence by remembering 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 element-wise.
- Finally, we can define the predictor in terms of a simple dot product between another weight vector and $\mathbf{h}$.
- 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

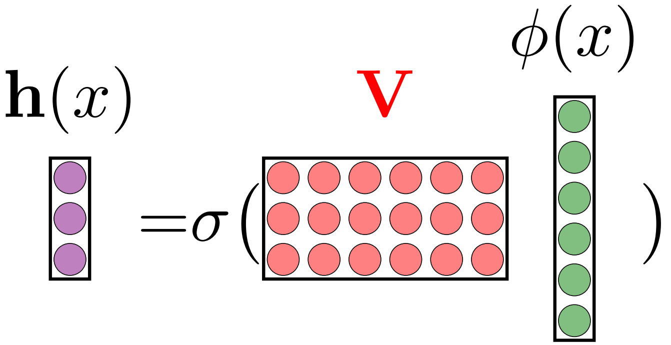


$$h_1(x) = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

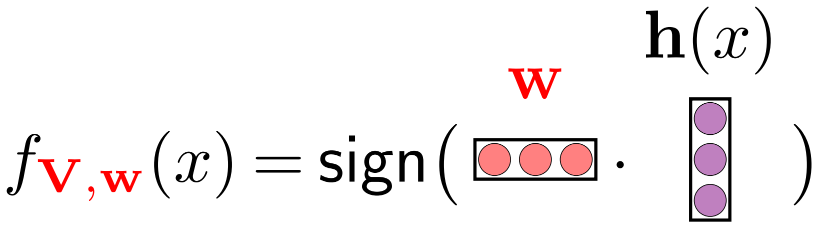
- Later we'll show how to perform learning using gradient descent, but we can anticipate one problem from the types of functions that we decided to use.
- For example, take a look at subproblem h_1. The gradient of $h_1(x)$ with respect to $\mathbf{v}_1$ is always zero because of the threshold function. We saw this issue before in the context of the zero-one loss function.
- To fix this, we replace the threshold function with what is called an **activation function** which has non-zero gradients, making it possible to optimize through
- Classically, neural networks approxiate the threshold funciton using the **logistic function** $\sigma(z)$, which is a softer version of the threshold function but has non-zero gradients everywhere.
- However, even though the gradients are non-zero, they can be quite small when $|z|$ is large. This phenomenon is known as saturation, and it still makes optimizing with the logistic function very difficult, and is one of the reasons why it took longer for neural networks to catch on.
- In 2012, Glorot et al. introduced the ReLU activation function, which stands for rectifying linear unit, and which is simply $\max(z, 0)$. This has the advantage that at least on the positive side, the gradient does not vanish; it does not go to zero even when z is large (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 de facto 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 expressive power of a neural network. This is because, if you remove the nonlinearity entirely – when you multiply $\mathbf{h}$ with our weight vector, the resulting function is fully linear. So, stacking together h_1 and our second layer would still be linear.
- Therefore, that there is a tension between wanting an activation function that is non-linear so that we can represent more complex functions but also wanting to have non-zero gradients so that we can optimize the weights. ReLU has served as a good compromise between the two.

Two-layer neural networks

Intermediate subproblems:



Predictor (classification):



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

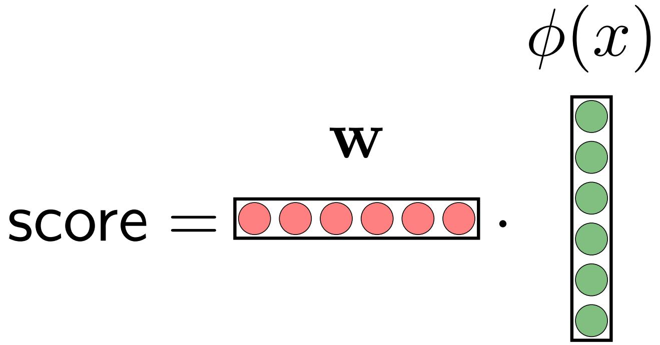
Hypothesis class:

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

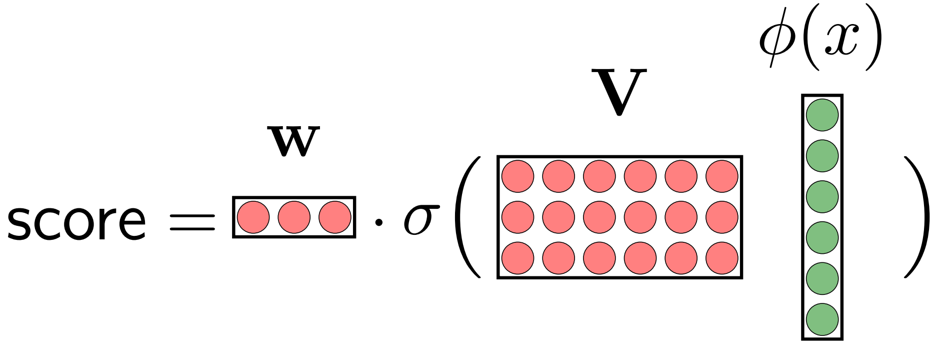
- Now we are finally ready to define the hypothesis class of two-layer neural networks.
- We start with a feature vector $\phi(x)$.
- We multiply it by a weight matrix $\mathbf{V}$ (whose rows can be interpreted as the weight vectors of the k intermediate subproblems.
- Then we apply the activation function σ to each of the k components to get the hidden representation $\mathbf{h}(x) \in \mathbb{R}^k$. You can think of this as the solution to the subproblems
- Now, 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 simply perform linear classification or regression – 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 then the set of all such predictors obtained by varying both the first-layer weight matrix $\mathbf{V}$ and the second-layer weight vector $\mathbf{w}$. So, unlike linear predictors, we now have two sets of weights.
- One way to think about this is that we have subproblems that are learning the features, and then we simply have a linear predictor on top of this learned representation. And, both the features and the linear predictor on top are being learned.

Deep neural networks

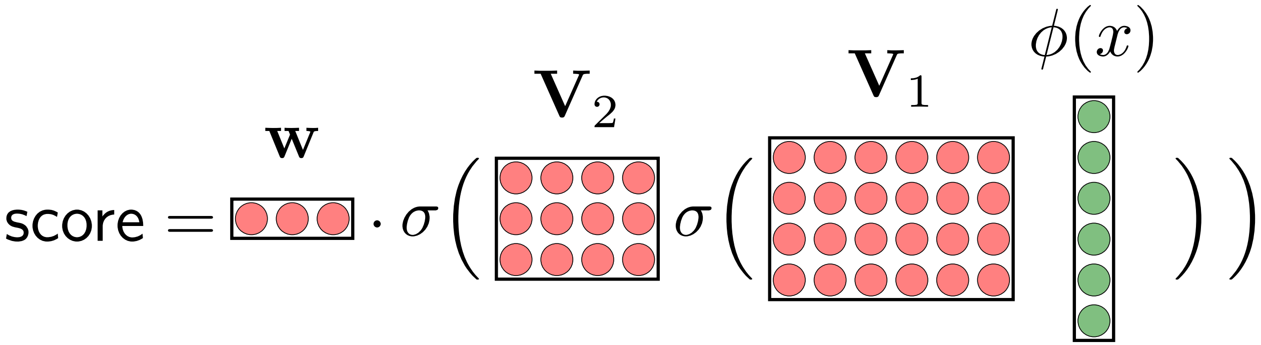
Linear predictor:



2-layer neural network:



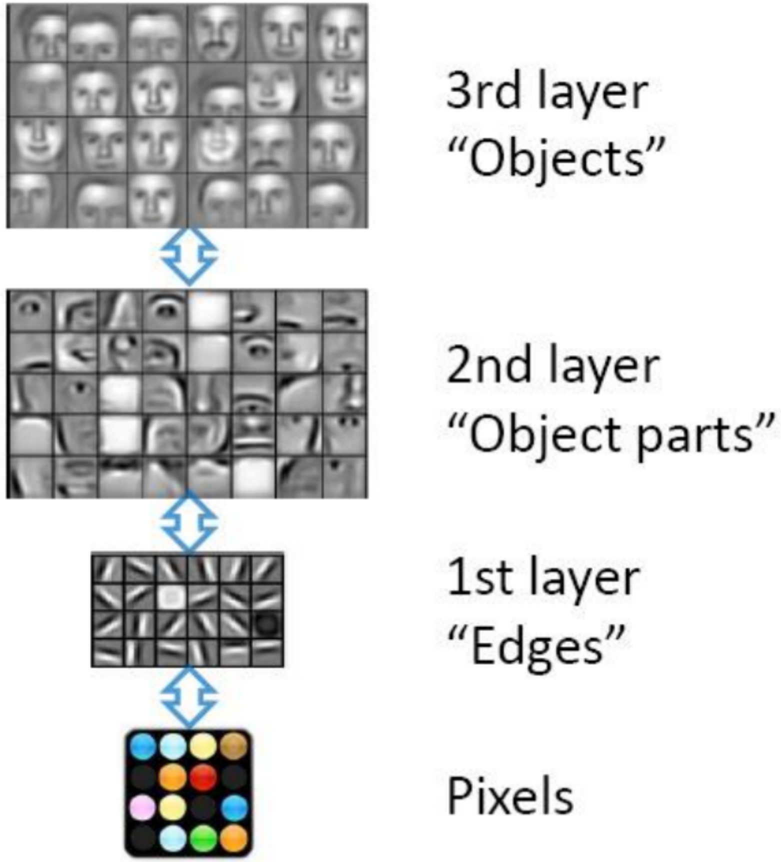
3-layer neural network:



- Now, can we go even 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.
- How does this work? As a warm-up, for a linear predictor (which you can think of as the precursor to a 2-layer neural network), 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 take our features and apply a linear layer $\mathbf{V}$ first, followed by a non-linearity σ (giving us learned features), and then take the dot product.
- To obtain a three-layer neural network, we repeat this process. We apply a linear layer and a non-linearity. This is the basic building block, and 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 or offset (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”

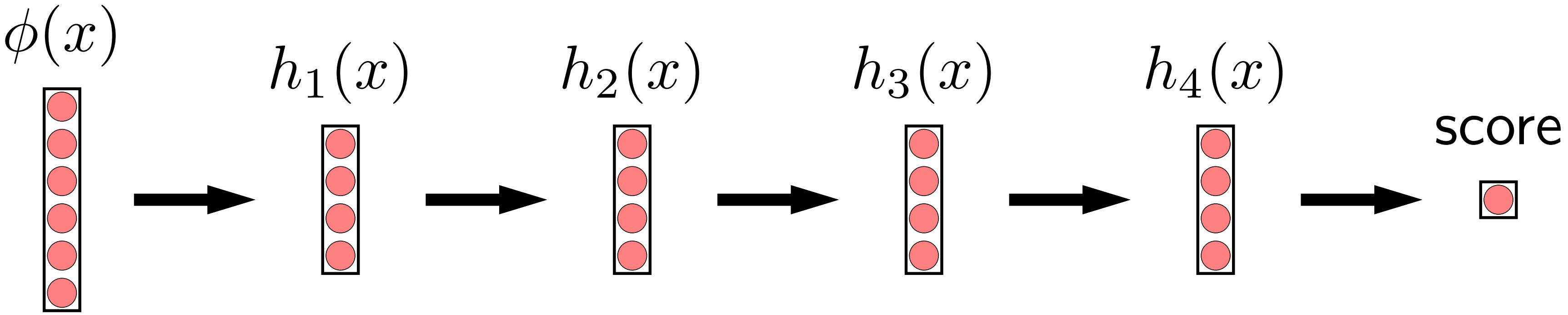
2nd layer
“Object parts”

1st layer
“Edges”

Pixels

- It can be difficult to understand what a sequence of these linear algebra operations means, and what (matrix multiply, non-linearity) operations buy you when they are stacked.
- 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 might detect object parts, and the third layer might detect entire objects or faces.
- Though we haven't talked about learning neural networks, it turns out that these kinds of abstractions are learned automatically when we train neural networks on real data and inspect what is learned. This kind of emergent phenomenon is also what makes neural networks pretty fascinating.

Why depth?

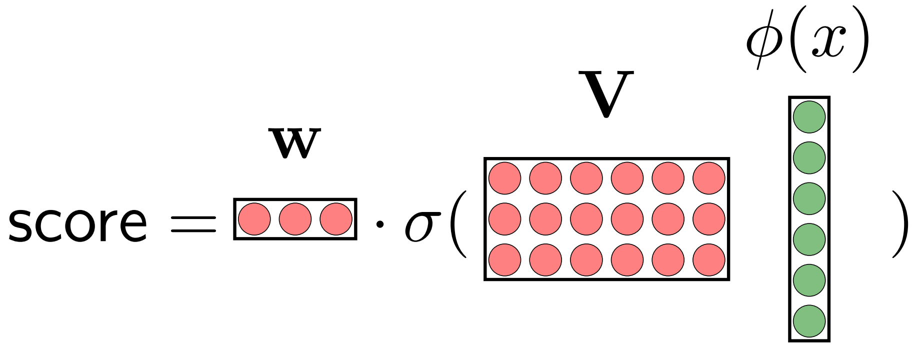


Intuitions:

- 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. They have achieved very impressive performance in multiple different domains, which canot be said for many other kinds of models.
- From a theoretical perspective, we have a very 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 circuites. 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 this section, 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 for all of the weights in deep neural networks.
- We are now going to discuss backpropagation which lets us compute gradients automatically given a specification of our model.

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. 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: in principle, you can simply apply the chain rule. But grinding through this complex expression by hand can be quite tedious, as you will need to apply the chain rule 6 or 7 times. What if 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

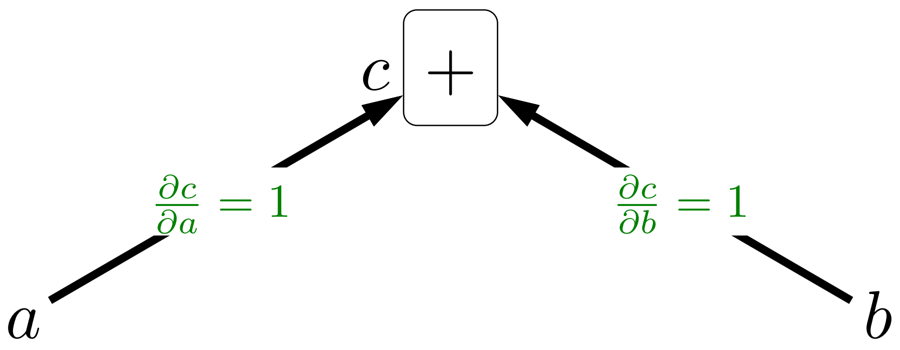
Purposes:

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

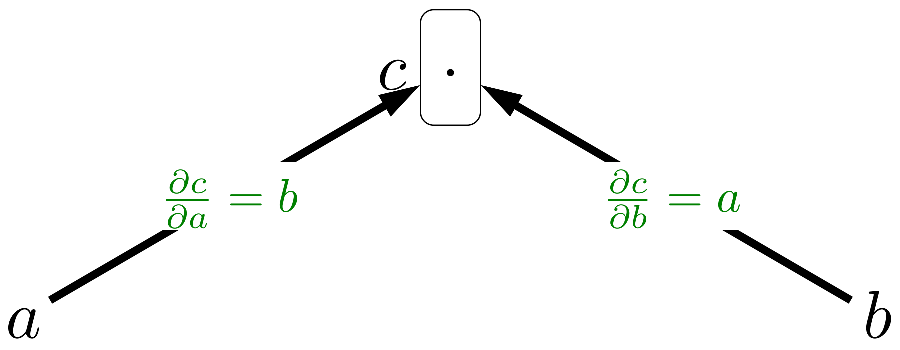
- This is where computation graphs are helpful.
- A computation graph is a directed acyclic graph that represents an arbitrary mathematical expression, such as the loss of a model. The root node of this graph represents the final expression, and the other nodes represent intermediate subexpressions.
- 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 using this idea.
- 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, and give you more intuition about how these networks are trained.

Functions as boxes

$$c = a + b$$


c
$+$
$\frac{\partial c}{\partial a} = 1$
$\frac{\partial c}{\partial b} = 1$
a
b

$$c = a \cdot b$$


c
$\cdot$
$\frac{\partial c}{\partial a} = b$
$\frac{\partial c}{\partial b} = a$
a
b

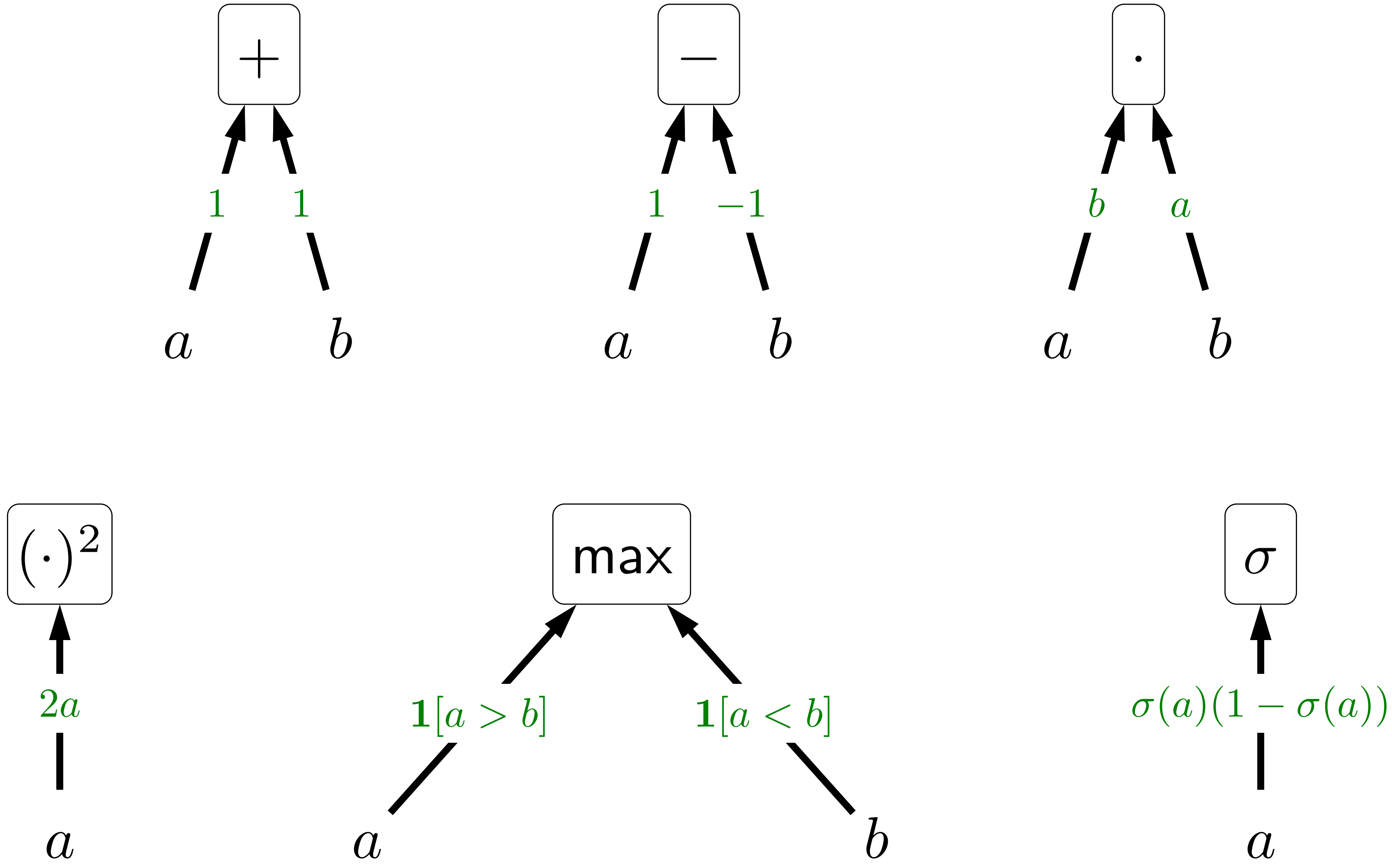
$$(a + \epsilon) + b = c + 1\epsilon$$
$$a + (b + \epsilon) = c + 1\epsilon$$

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

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

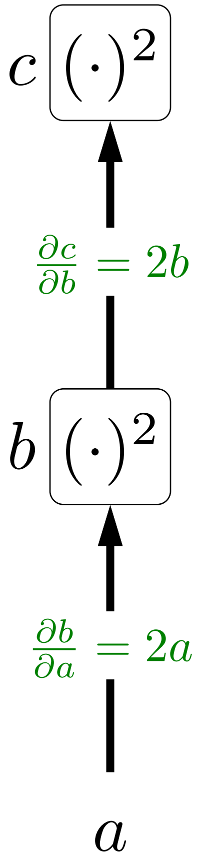
- The first conceptual step is to think of functions as nodes in a graph that take a set of inputs and produces an output.
- For example, take $c = a + b$. The key question that we want to ask: if we change a by a small amount ϵ, how much does the output c change? In this case, the output c is also perturbed by 1ϵ, so the gradient (or the partial derivative) is 1. We put this gradient on the edge of the graph.
- 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 visual way.
- We already saw addition, which has gradients of 1 with respect to a and 1 with respect to b, and subtraction is similar but with a -1 for the second term. We also saw multiplication, where the gradient with respect to a is b and vice versa. Squaring, by applying the chain rule, gives us 2 times the input which is a, so 2a.
- For the max function that takes the max of a and b, changing a only impacts the max iff $a > b$; and analogously for b.
- For the logistic function $\sigma(a) = \frac{1}{1+e^{-a}}$, a bit of algebra produces the gradient. You can chack this by hand if you would like. You can also check that the gradient is zero when $|a| \to \infty$.
- While this is hopefully all quite simple and straightforward, it turns out that these building blocks are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter when building and computing gradients for neural networks.

Function composition



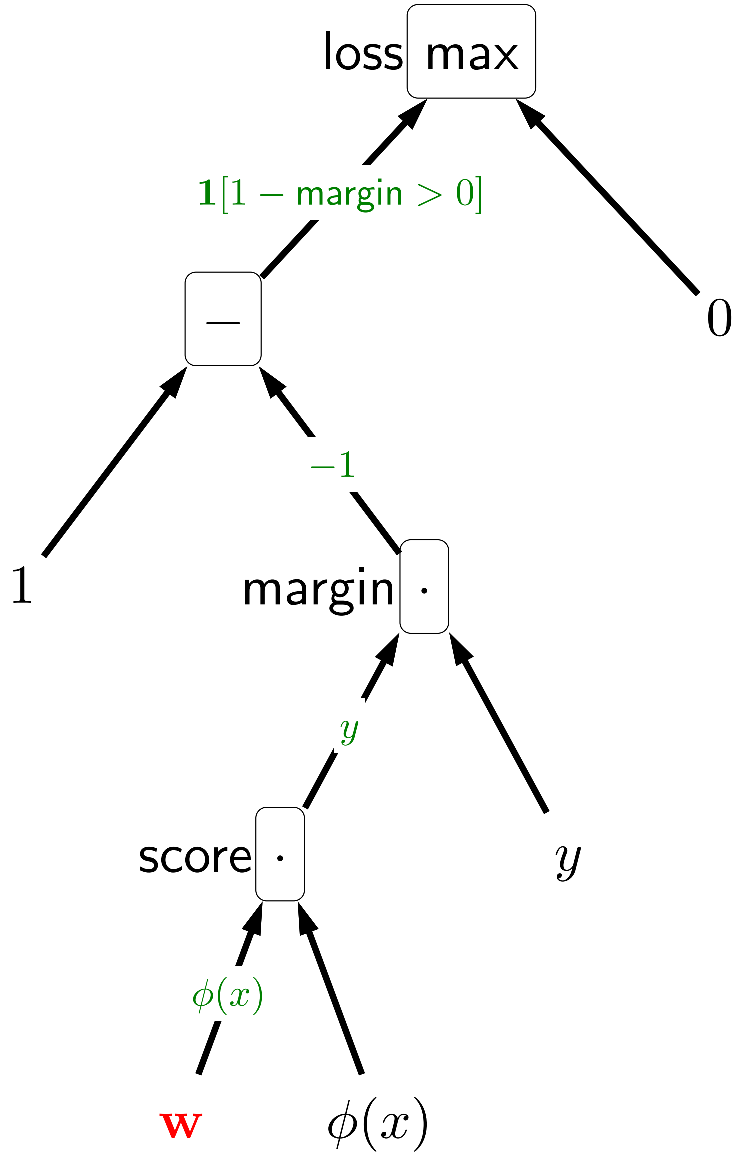
c
$(\cdot)^2$
$\frac{\partial c}{\partial b} = 2b$
b
$(\cdot)^2$
$\frac{\partial b}{\partial a} = 2a$
a

Chain rule:

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

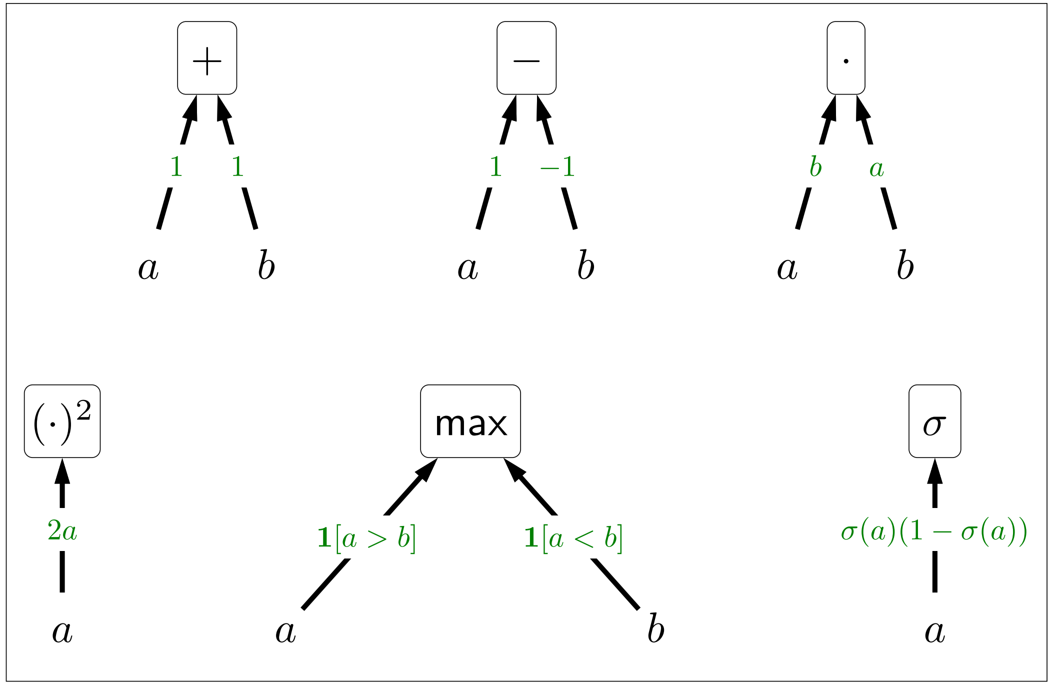
- Given these building blocks, we can now put them together to create more complex functions.
- Consider applying some function (e.g., squared) to a to get b, and then applying some other function (e.g., squared) to get c.
- What is the gradient of c with respect to a?
- We know from our building blocks the gradients on the edges.
- So how do we combine them? The final answer is given by the **chain rule** from calculus. We 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 how to compute gradients for much more complex functions.

Linear classification with hinge loss



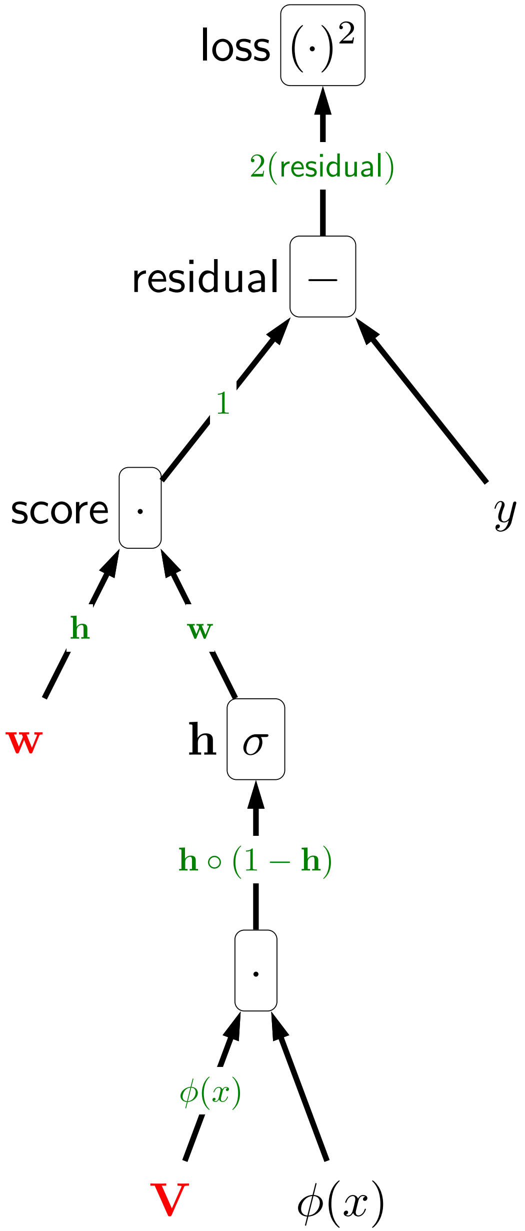
$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x) y, 0\}$$

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



- 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 from the bottom up. At the leaves are the inputs and the constants, and the root is the full loss. 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 $\mathbf{w}$, which is what we want to take gradients with respect to. Then, the central question is how small perturbations in $\mathbf{w}$ affect the loss.
- We can do this by examining each edge from the path from $\mathbf{w}$ to loss, and compute the gradient using our handy reference of building blocks.
- Then, the gradient is the product of the edge-wise gradients from $\mathbf{w}$ to the loss output.
- Thus, we see that instead of doing a lot of new algebraic computations, we can simply compose each of the building blocks that we had already computed before.

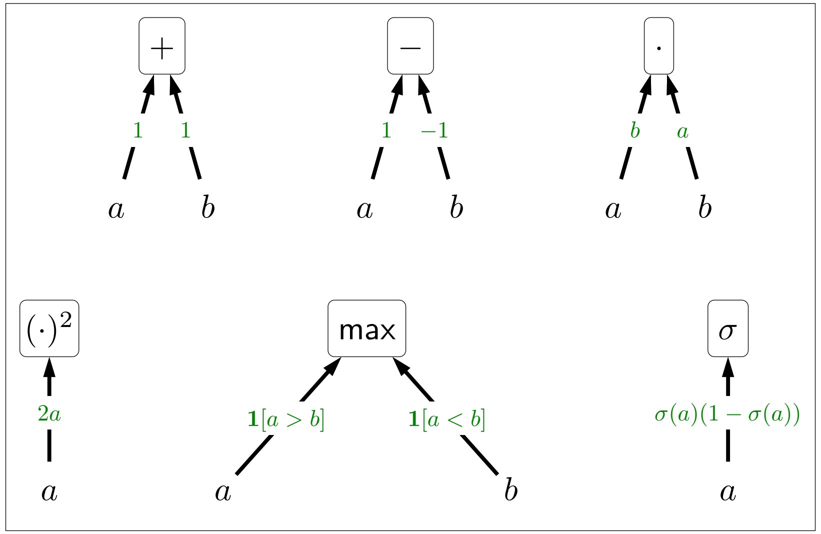
Two-layer neural networks



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

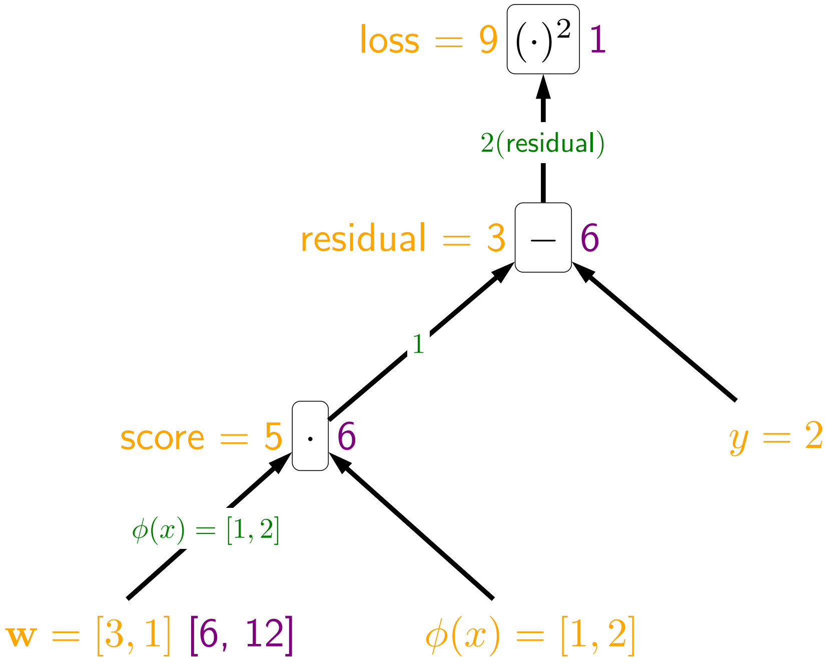
$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

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



- We now finally turn to neural networks, and the idea is essentially the same using the same building blocks.
- Specifically, consider a two-layer neural network driving the squared loss.
- Let's build the computation graph bottom up.
- Now we need to take the gradient of the loss with respect to our weights $\mathbf{w}$ 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.
- For $\mathbf{w}$, the gradient computation is relatively simple. What about $\mathbf{V}$? There are some technicalities when dealing with vectors: I have been talking about gradients this whole time, but when a function is vector valued, what we actually want is the Jacobian Gradients are a special case when the function's output is a single scalar value.
- 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. You can verify this by hand by taking the derivative for each entry of the vector and looking at the resulting vector of derivatives.
- 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 one check 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. Checking that the dimensions of your gradiente match the weights is always a good sanity check.
- This computation graph also highlights the modularity of the 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)

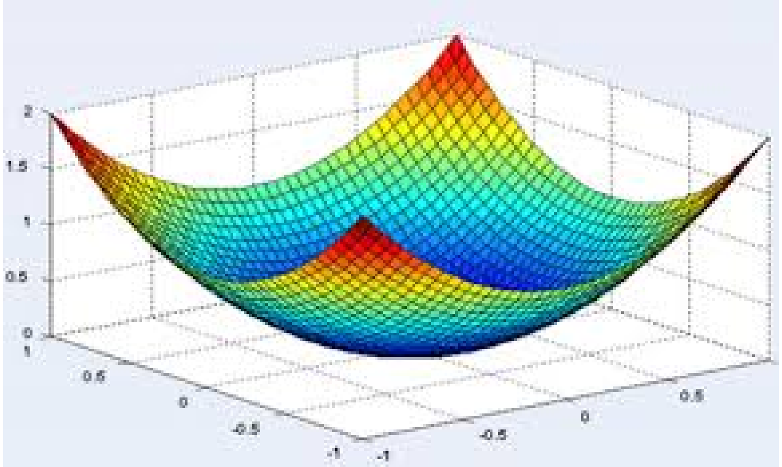
Backward pass: compute each g_i (from root to leaves)

- 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, which 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 take all of the inputs and compute a **forward value** f_i for each node, coresponding to the evaluation of that subexpression.
- Let's work through the example. Our inputs are phi, y, and w, which we can use to populate our leaf nodes. Then we compute every subexpression that we can: first the score by taking the dot product between the features and the weights, then the residual using the score and y, finally the loss using the residual. This was the forward phase.
- After the forward pass is the backward phase. In the backward phase, we compute a **backward value** g_i for each node, which is the gradient of the root (i.e. the loss) with respect to that node. Because of the chain rule, this gradient is 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. We will always start with the value of 1 because the gradient of the loss with respect to itself is 1. To get the gradient w.r.t. the residual, we multiple 1 by 2 times the residual, which gives us 6. Now we continue working backwards: since the next edge has a gradient of 1 we get a backwards value of 6. Finally, we get to the weights (which is exactly what we want to compute the gradient with respect to) and multiply 6 with the features to get our gradient update.
- Note that both f_i and g_i can either be scalars, vectors, or matrices, but f_i and g_i have the same dimensionality as each other.

A note on optimization

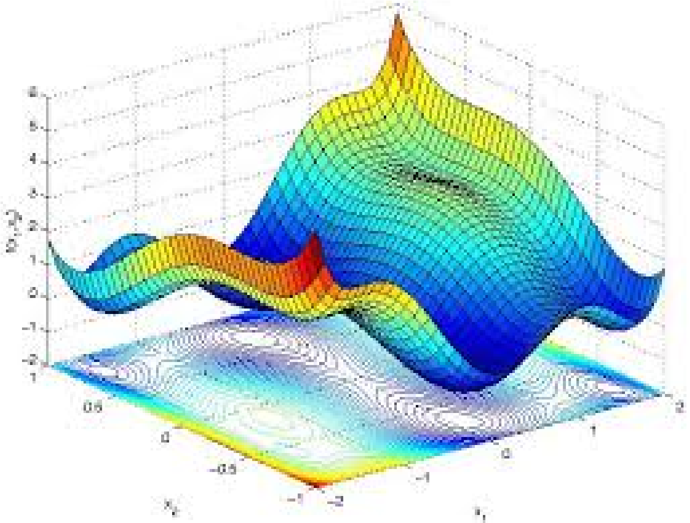
$$\min_{\mathbf{V},\mathbf{w}} \mathsf{TrainLoss}(\mathbf{V},\mathbf{w})$$

Linear predictors



(convex)

Neural networks

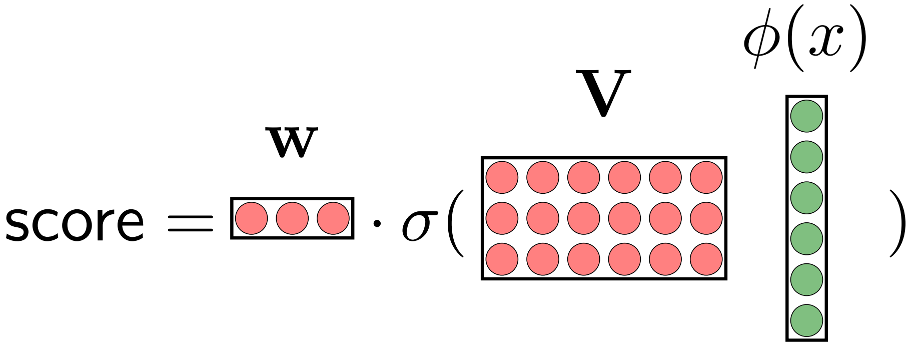


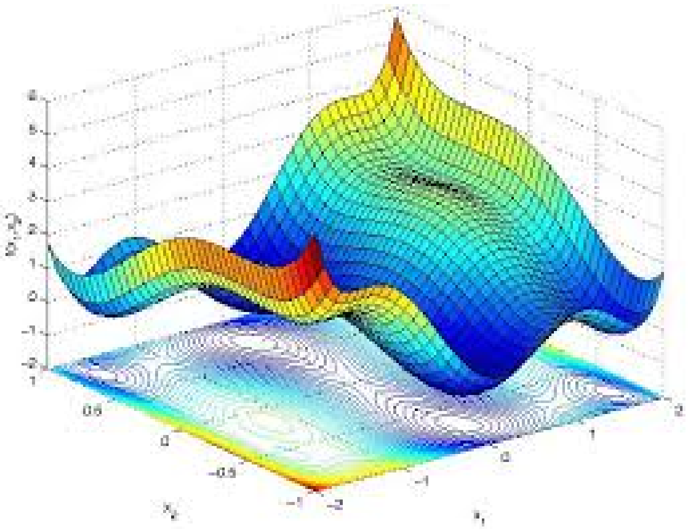
(non-convex)

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), $\text{TrainLoss}(\mathbf{w})$ is a convex function (like the picture on the left), which means that SGD (with an appropriate step size) is theoretically guaranteed to converge to the global optimum.
- However, for neural networks, $\text{TrainLoss}(\mathbf{V}, \mathbf{w})$ is typically a non-convex function of the weights 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 such as proper choice of initialization and step size. The gap between theory and practice is not particularly well understood and an active area of research.

How to train neural networks



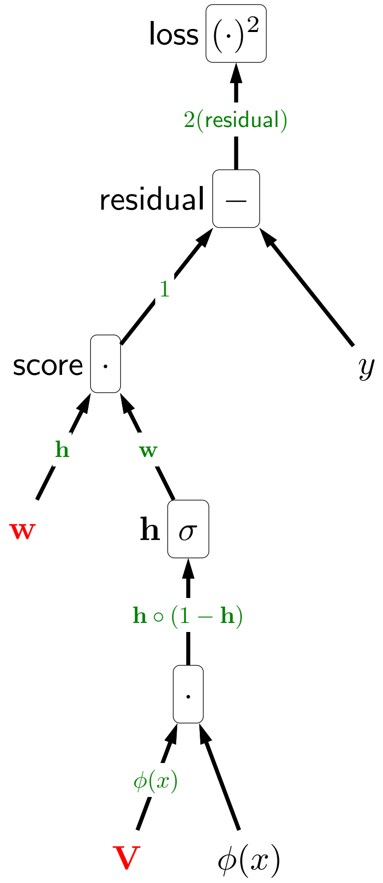


- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

Don't let gradients vanish or explode!

- Training a neural network is very much like driving a manual transmission car. In practice, there are some "tricks" that are needed to make things work properly. I'll just 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 your weights at zero or else all of the gradients will all be zero. Instead, you want to initialize with a small amount of random noise, which breaks ties and allows the optimization to proceed.
- It is common to use overparameterized neural networks, ones with more hidden units (k) and depth than what is needed, because then there are more "chances" that some of them will pick out on the right signal,
- There are small but important extensions of stochastic gradient descent that allow the step size to be tuned per weight, which allows the optimizer to behave better.
- 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 last 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 by looking at internal nodes.
- The generality of computation graphs and backpropagation makes it possible to iterate very quickly on new types of models and loss functions. This is one of the reasons why deep learning has grown quickly and tackled so many different problems.

Lecture Recap

- **Nonlinear features**: nonlinear predictors with linear machinery
- **Feature templates**: organization of features, including sparse features
- **Neural networks**: learn nonlinear features through composition of linear classifiers
- **Backpropagation**: general-purpose method for computing gradients using computation graph

- To recap the entire lecture, we saw two different ways to construct more expressive predictors that are not simply linear functions.
- Nonlinear features gave us a way to construct nonlinear predictors with all of the machinery of linear regression and classification, essentially by preprocessing the inputs.
- Feature templates provided an systematic way to define such features, and we saw how dictionaries can be used to efficiently represent sparse features.
- Then, we saw how we can learn features more automatically using neural networks that essentially stack and compose linear classifiers – interleaving linear weight matrices and nonlinear activation functions.
- Finally, we saw that we can compute gradients of neural networks with backpropagation by taking a forward and backward pass through the computation graph.