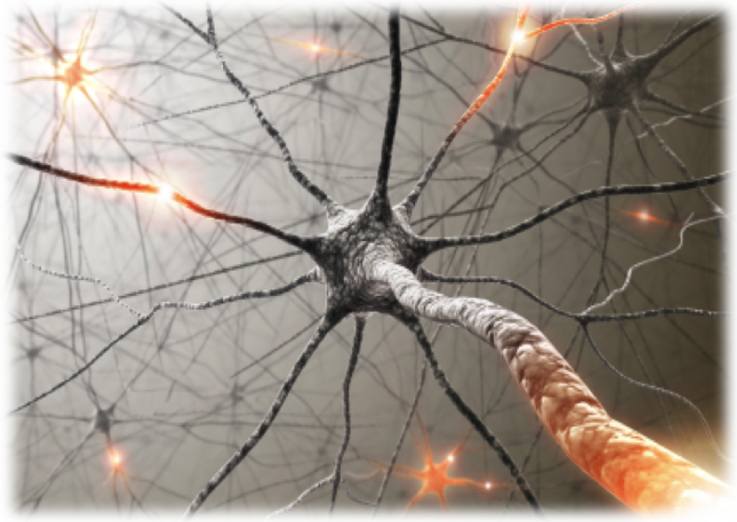
Lecture 4: Machine Learning 3



Roadmap

Backpropagation

K-means

Generalization

Best practices

Summary of Machine Learning

- In this module, I'll discuss **backpropagation**, an algorithm to automatically compute gradients.
- It is generally associated with training neural networks, but actually it is much more general and applies to any function.

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:

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

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

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

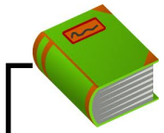
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{w})$$

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 between 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 the respective weight vector/matrix by subtracting the gradient times a step size η.
- 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.

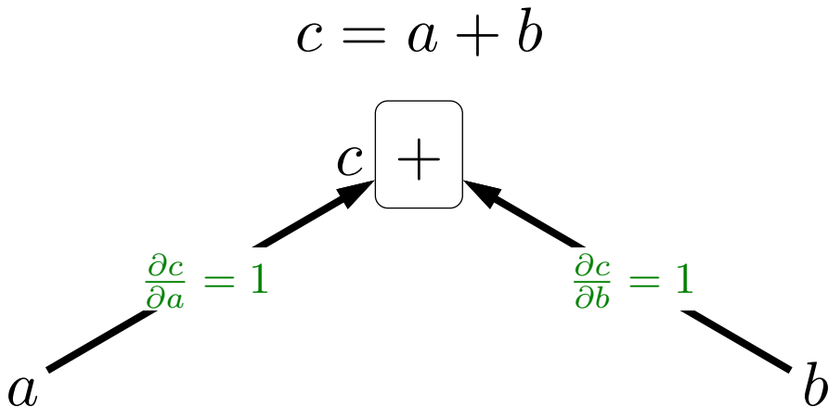
Upshot: compute gradients via general **backpropagation** algorithm

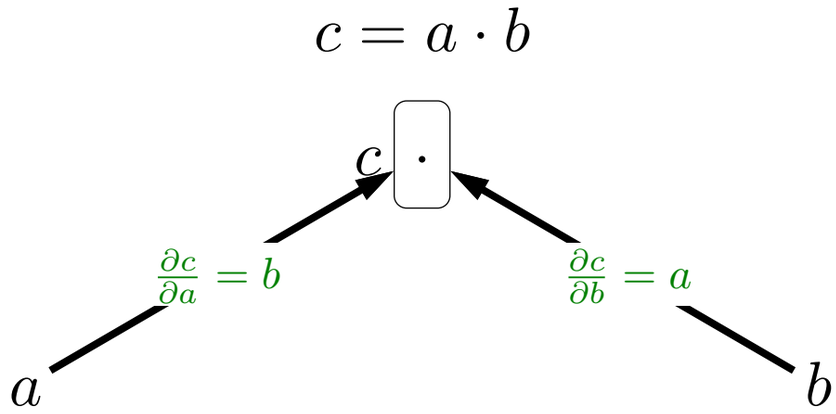
Purposes:

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

- Enter computation graphs, which will rescue us.
- 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.

Functions as boxes



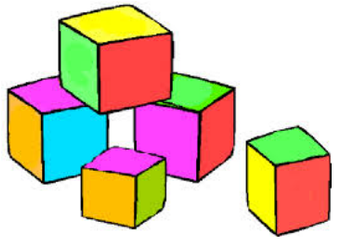


$$(a + \epsilon) + b = c + 1\epsilon$$
$$a + (b + \epsilon) = c + 1\epsilon$$

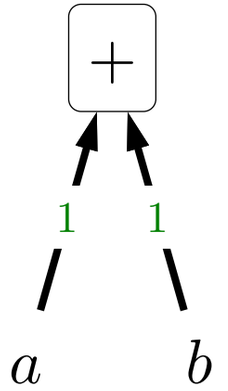
$$(a + \epsilon)b = c + b\epsilon$$
$$a(b + \epsilon) = c + a\epsilon$$

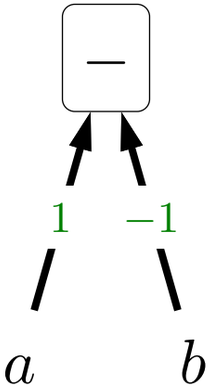
Gradients: how much does c change if a or b changes?

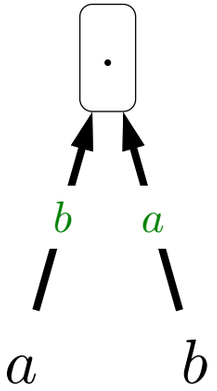
- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output.
- For example, take $c = a + b$. The key question is: if we perturb a by a small amount ϵ, how much does the output c change? In this case, the output c is also perturbed by 1ϵ, so the gradient (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.

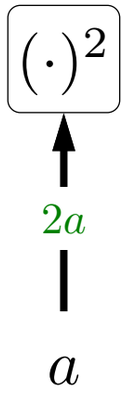


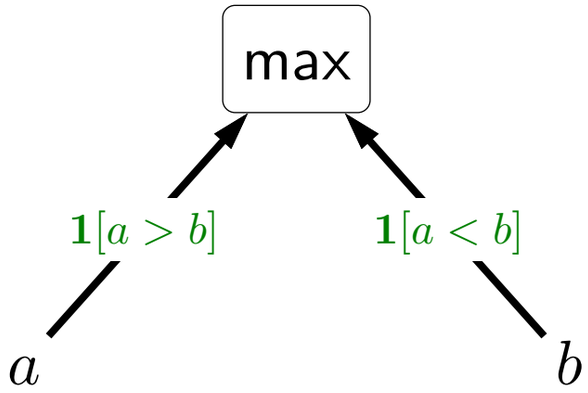
Basic building blocks

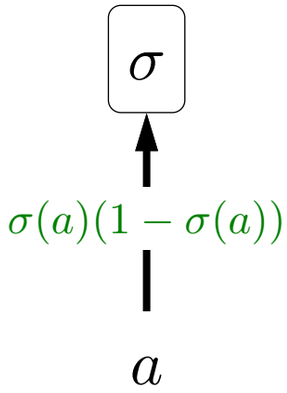






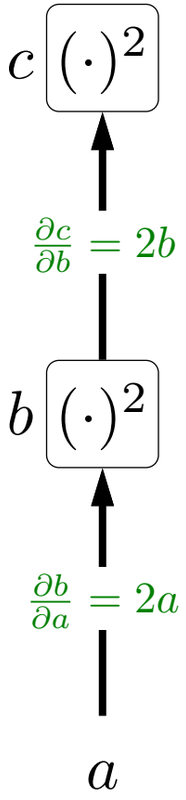






- Here are some more examples of simple functions and their gradients. Let's walk through them together.
- These should be familiar from basic calculus. All we've done is present them in a visually more intuitive way.
- For the max function, changing a only impacts the max iff $a > b$; and analogously 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

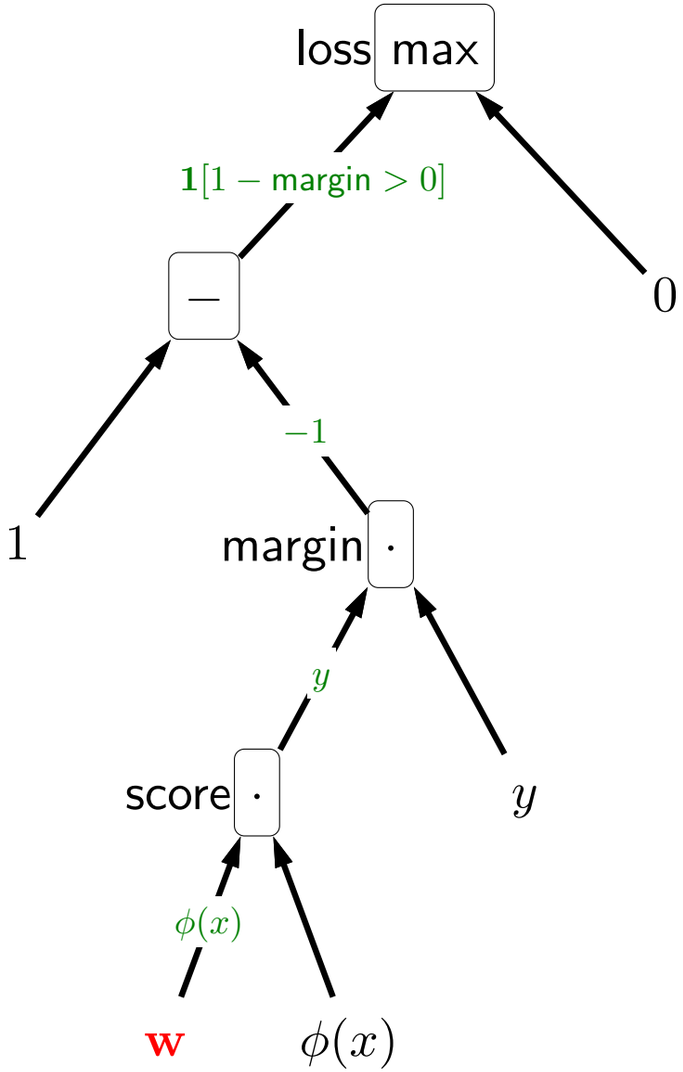


Chain rule:

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

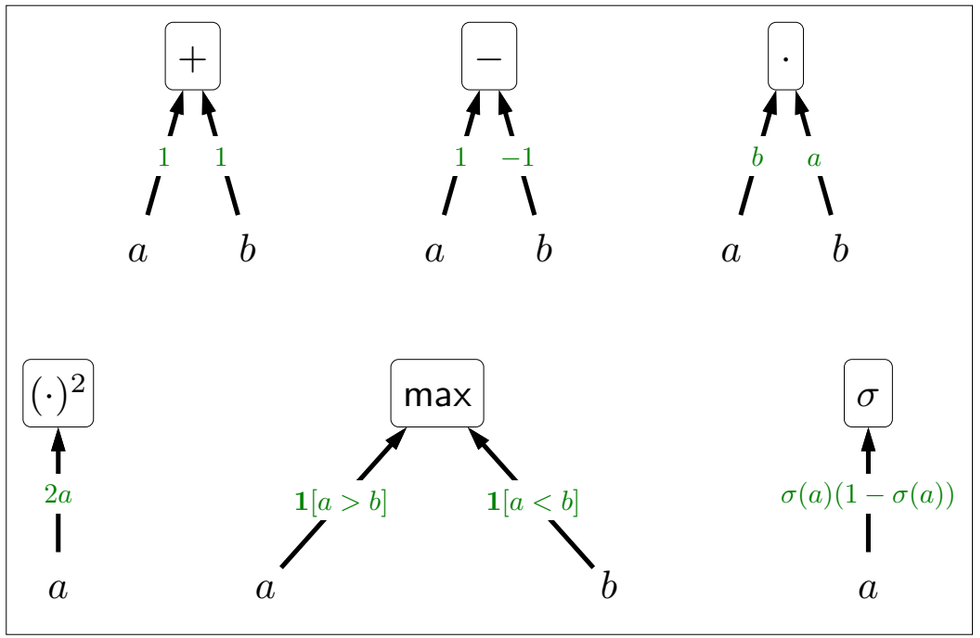
- Given these building blocks, we can now put them together to create more complex functions.
- Consider applying some function (e.g., squared) to a to get b, and then applying some other function (e.g., squared) to get c.
- What is the gradient of c with respect to a?
- We know from our building blocks 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



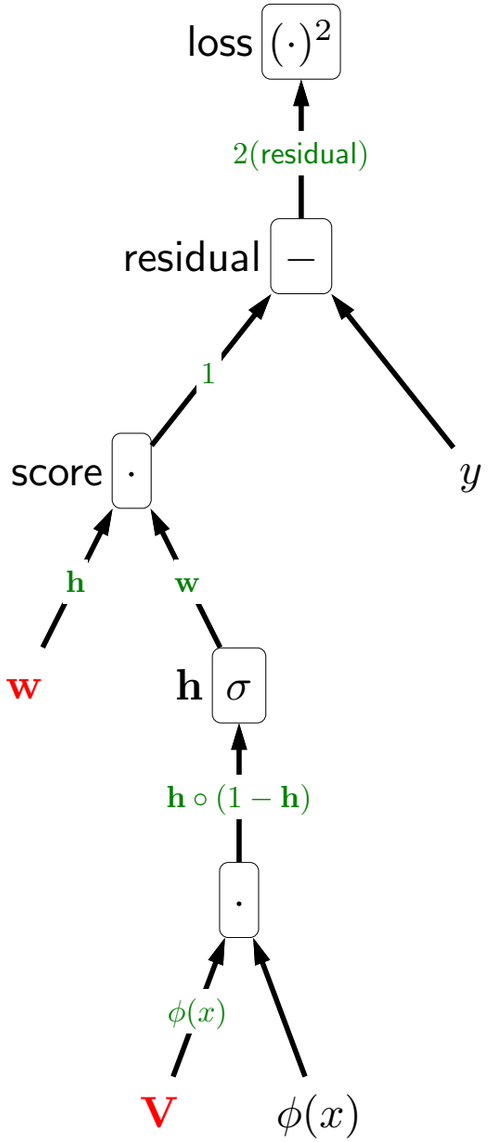
$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

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



- 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.

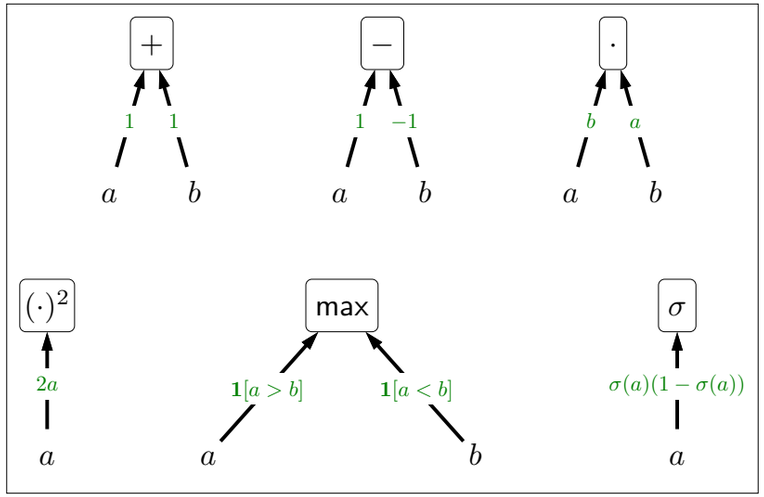
Two-layer neural networks



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

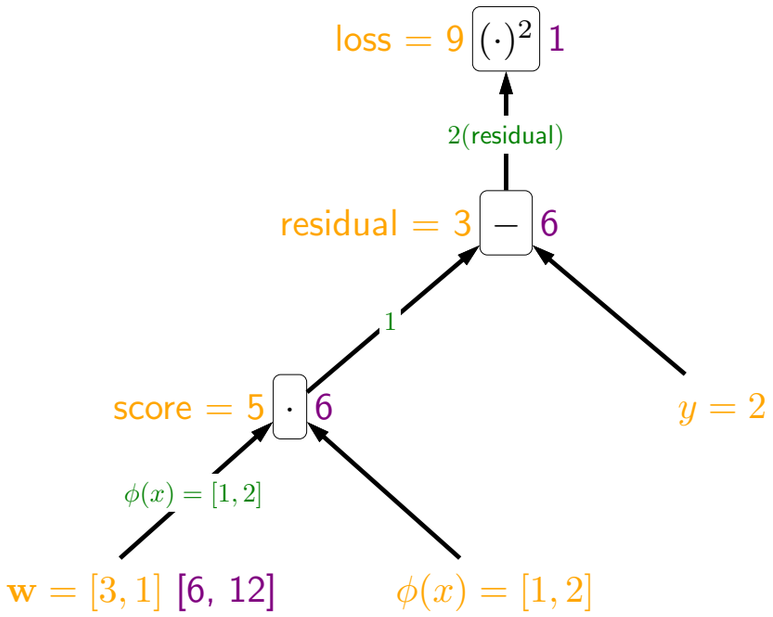
$$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = 2(\text{residual})\mathbf{h}$$

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



- 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, there is a transpose for the gradient expression with respect to $\mathbf{V}$ and not $\mathbf{w}$ because we are taking $\mathbf{V}\phi(x)$, while taking $\mathbf{w} \cdot \mathbf{h} = \mathbf{w}^\top \mathbf{h}$.
- 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

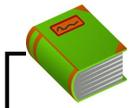


$$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

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

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)

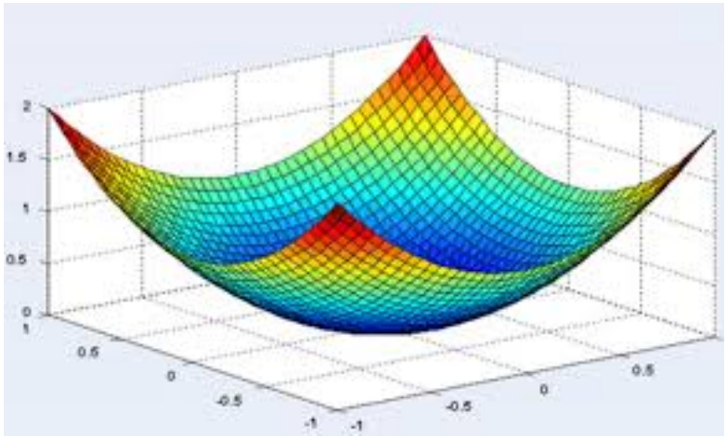
Backward pass: compute each g_i (from root to leaves)

- 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

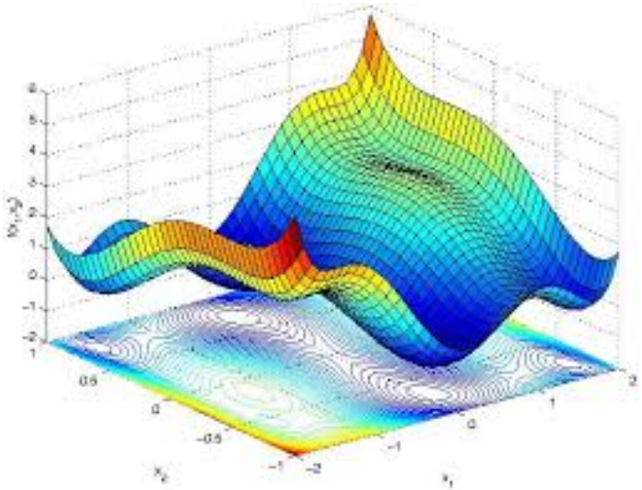
$$\min_{\mathbf{V},\mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

Linear predictors



(convex)

Neural networks

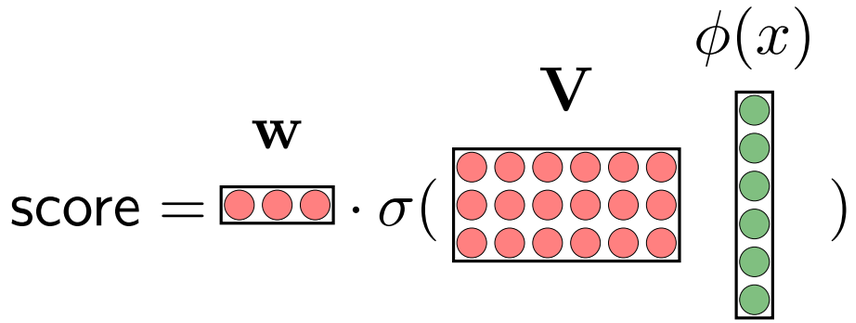


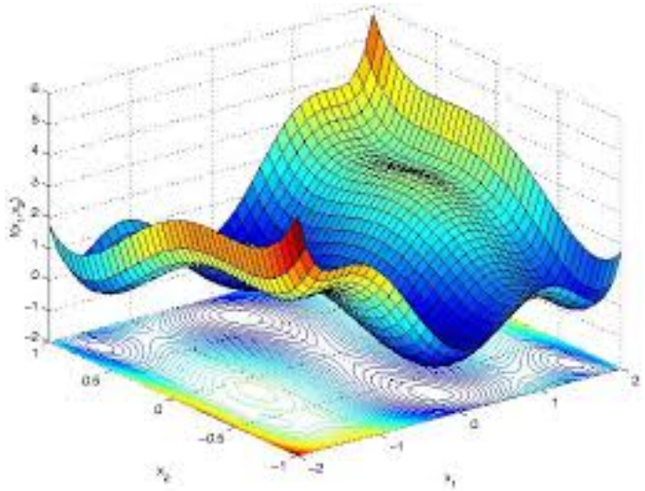
(non-convex)

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 appropriately 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



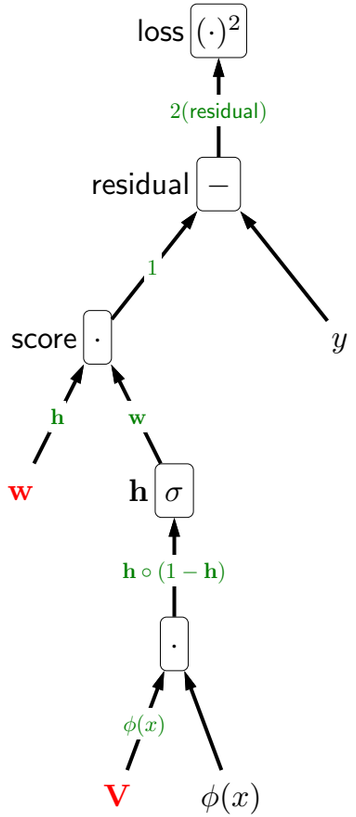


- Careful initialization (random noise, pre-training)
- Overparameterization (more hidden units than needed)
- Adaptive step sizes (AdaGrad, Adam)

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 module is the idea of a **computation graph**, allows us to represent arbitrary mathematical expressions, which can just be built out of 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 and opens up a new paradigm for model development: **differential programming**.

Roadmap

Backpropagation

K-means

Generalization

Best practices

Summary of Machine Learning

- In this module, we'll talk about **K-means**, a simple algorithm for clustering, a form of unsupervised learning.

[Brown et al, 1992]

Word clustering

Input: raw text (100 million words of news articles)...

Output:

Cluster 1: Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays
Cluster 2: June March July April January December October November September August
Cluster 3: water gas coal liquid acid sand carbon steam shale iron
Cluster 4: great big vast sudden mere sheer gigantic lifelong scant colossal
Cluster 5: man woman boy girl lawyer doctor guy farmer teacher citizen
Cluster 6: American Indian European Japanese German African Catholic Israeli Italian Arab
Cluster 7: pressure temperature permeability density porosity stress velocity viscosity gravity tension
Cluster 8: mother wife father son husband brother daughter sister boss uncle
Cluster 9: machine device controller processor CPU printer spindle subsystem compiler plotter
Cluster 10: John George James Bob Robert Paul William Jim David Mike
Cluster 11: anyone someone anybody somebody
Cluster 12: feet miles pounds degrees inches barrels tons acres meters bytes
Cluster 13: director chief professor commissioner commander treasurer founder superintendent dean custodian
Cluster 14: had hadn't hath would've could've should've must've might've
Cluster 15: head body hands eyes voice arm seat eye hair mouth

- Here is a classic example of clustering from the NLP literature, called Brown clustering. This was the unsupervised learning method of choice before word vectors.
- The input to the algorithm is simply raw text, and the output is a clustering of the words.
- The first cluster more or less represents days of the week, the second is months, the third is natural resources, and so on.
- It is important to note that no one told the algorithm what days of the week were or months or family relations. The clustering algorithm discovered this structure automatically.
- On a personal note, Brown clustering was actually my first experience that got me to pursue research in NLP. Seeing the results of unsupervised learning when it works was just magical. And of course today, we're seeing even more strongly the potential of unsupervised learning with neural language models such as BERT and GPT-3.

Classification (supervised learning)

training data $\mathcal{D}_{\text{train}}$

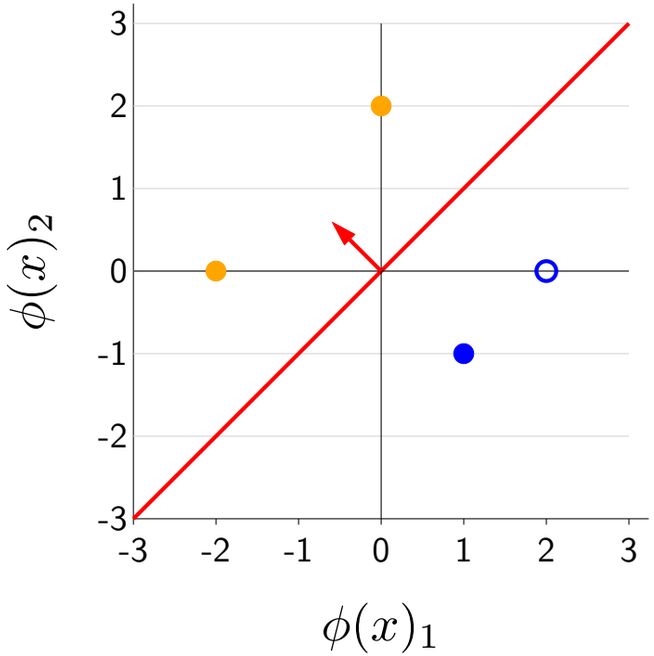
$\phi(x)_1$	$\phi(x)_2$	y
0	2	1
-2	0	1
1	-1	-1

learning algorithm

[2, 0]

f classifier

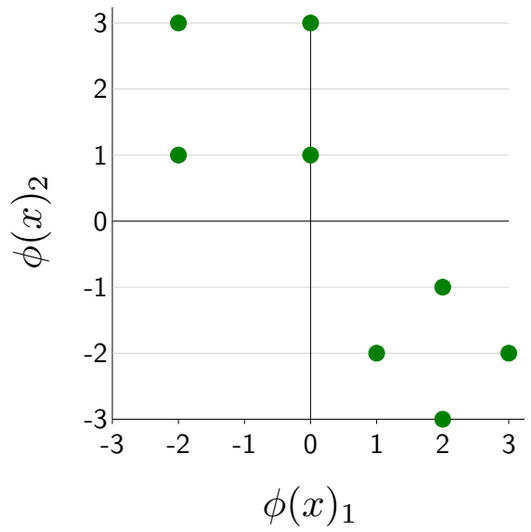
-1



Labeled data is expensive to obtain

- I want to contrast unsupervised learning with supervised learning.
- Recall that in classification you're given a set of **labeled** training examples.
- A learning algorithm produces a classifier that can classify new points.
- Note that we're now plotting the (two-dimensional) feature vector rather than the raw input, since the learning algorithms only depend on the feature vectors.
- However, the main challenge with supervised learning is that it can be expensive to collect the labels for data.

Clustering (unsupervised learning)



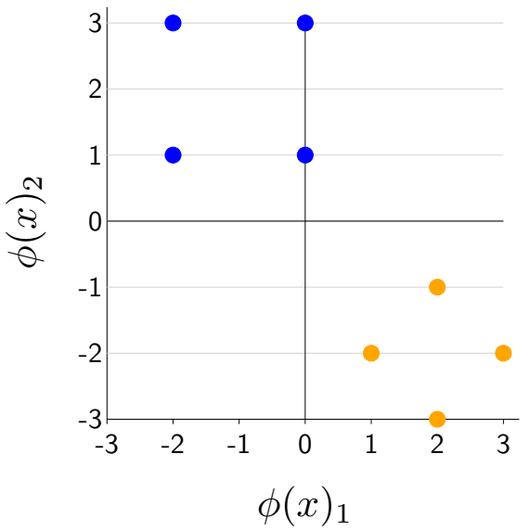
training data $\mathcal{D}_{\text{train}}$

$\phi(x)_1$	$\phi(x)_2$
-2	1
0	1
-2	3
0	3
2	-1
1	-2
2	-3
3	-2

learning algorithm

assignments

z
1
1
1
1
2
2
2
2

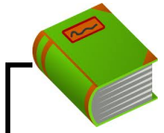


Intuition: Want to assign nearby points to same cluster

Unlabeled data is very cheap to obtain

- In contrast, in clustering, you are only given **unlabeled** training examples.
- Our goal is to assign each point to a cluster. In this case, there are two clusters, 1 (blue) and 2 (orange).
- Intuitively, nearby points should be assigned to the same cluster.
- The advantage of unsupervised learning is that unlabeled data is often very cheap and almost free to obtain, especially text or images on the web.

Clustering task



Definition: clustering

Input: training points

$$\mathcal{D}_{\text{train}} = [x_1, \ldots, x_n]$$

Output: assignment of each point to a cluster

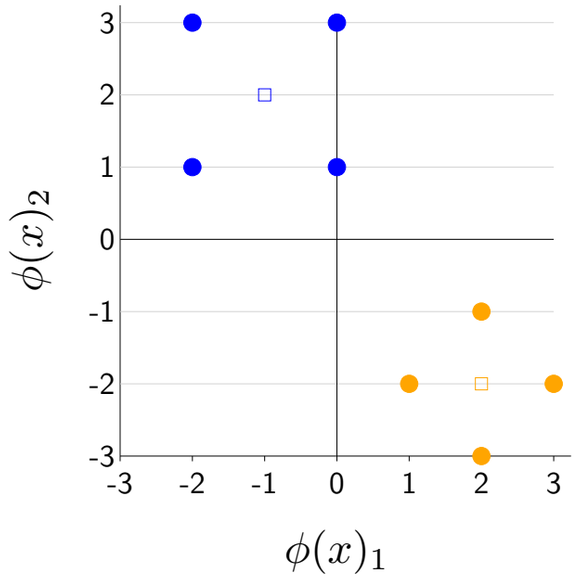
$$\mathbf{z} = [z_1, \ldots, z_n] \text{ where } z_i \in \{1, \ldots, K\}$$

- Formally, the task of clustering is to take a set of points as input and return a partitioning of the points into K clusters.
- We will represent the partitioning using an **assignment vector** $\mathbf{z} = [z_1, \ldots, z_n]$.
- For each i, $z_i \in \{1, \ldots, K\}$ specifies which of the K clusters point i is assigned to.

Centroids

Each cluster $k = 1, \ldots, K$ is represented by a **centroid** $\mu_k \in \mathbb{R}^d$

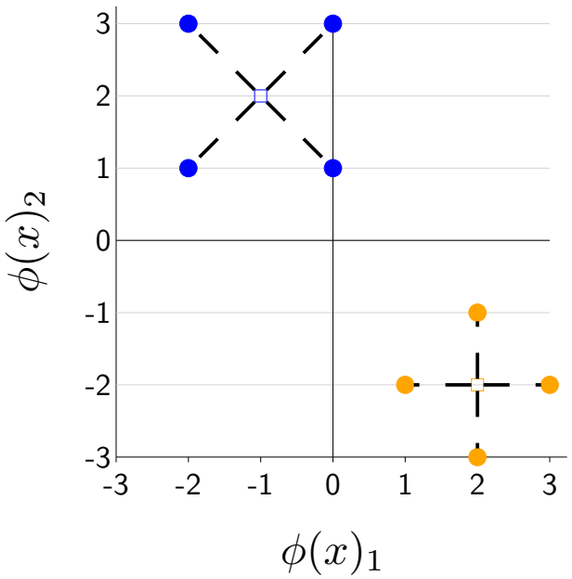
$$\boldsymbol{\mu} = [\mu_1, \ldots, \mu_K]$$



Intuition: want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i}

- What makes a cluster? The key assumption is that each cluster k is represented by a **centroid** μ_k.
- Now the intuition is that we want each point $\phi(x_i)$ to be close to its assigned centroid μ_{z_i}.

K-means objective

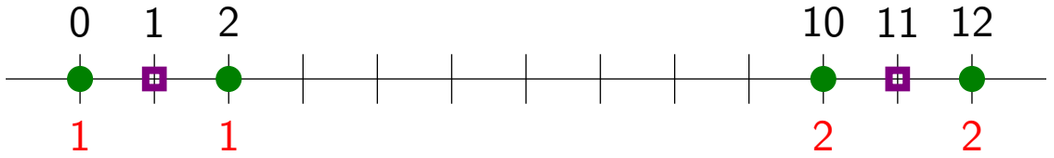


$$\text{Loss}_{\text{kmeans}}(\mathbf{z}, \boldsymbol{\mu}) = \sum_{i=1}^{n} \|\phi(x_i) - \mu_{z_i}\|^2$$

$$\min_{\mathbf{z}} \min_{\boldsymbol{\mu}} \text{Loss}_{\text{kmeans}}(\mathbf{z}, \mu)$$

- To formalize this, we define the K-means objective (distinct from the K-means algorithm).
- The variables are the assignments $\mathbf{z}$ and centroids $\boldsymbol{\mu}$.
- We examine the squared distance (dashed lines) from a point $\phi(x_i)$ to the centroid of its assigned cluster μ_{z_i}. Summing over all these squared distances gives the K-means objective.
- This loss can be interpreted as a reconstruction loss: imagine replacing each data point by its assigned centroid. Then the objective captures how lossy this compression was.
- Now our goal is to minimize the K-means loss.

Alternating minimization from optimum



If know centroids $\mu_1 = 1$, $\mu_2 = 11$:

$$z_1 = \arg\min\{(0-1)^2, (0-11)^2\} = 1$$
$$z_2 = \arg\min\{(2-1)^2, (2-11)^2\} = 1$$
$$z_3 = \arg\min\{(10-1)^2, (10-11)^2\} = 2$$
$$z_4 = \arg\min\{(12-1)^2, (12-11)^2\} = 2$$

If know assignments $z_1 = z_2 = 1$, $z_3 = z_4 = 2$:

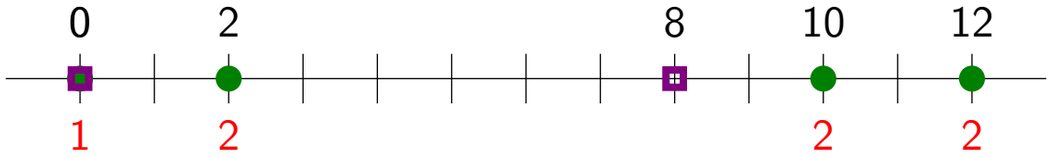
$$\mu_1 = \arg\min_\mu (0-\mu)^2 + (2-\mu)^2 = 1$$
$$\mu_2 = \arg\min_\mu (10-\mu)^2 + (12-\mu)^2 = 11$$

- Before we present the K-means algorithm, let us form some intuitions.
- Consider the following one-dimensional clustering problem with 4 points. Intuitively there are two clusters.
- Suppose we know the centroids. Then for each point the assignment that minimizes the K-means loss is the closer of the two centroids.
- Suppose we know the assignments. Then for each cluster, we average the points that are assigned to that cluster.

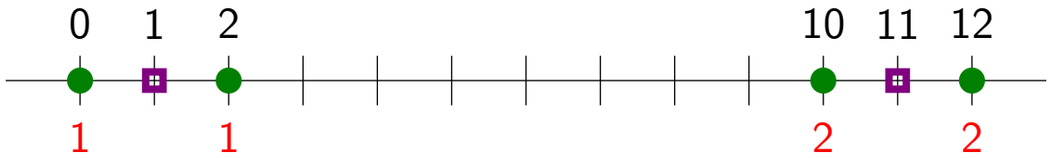
Alternating minimization from random initialization

Initialize μ:

Iteration 1:



Iteration 2:



Converged.

- But of course we don't know either the centroids or assignments.
- So we simply start with an arbitrary setting of the centroids.
- Then alternate between choosing the best assignments given the centroids, and choosing the best centroids given the assignments.
- This is the K-means algorithm.

K-means algorithm

Algorithm: K-means

Initialize $\boldsymbol{\mu} = [\mu_1, \ldots, \mu_K]$ randomly.

For $t = 1, \ldots, T$:

Step 1: set assignments $\mathbf{z}$ given $\boldsymbol{\mu}$

For each point $i = 1, \ldots, n$:

$$z_i \leftarrow \arg\min_{k=1,\ldots,K} \|\phi(x_i) - \mu_k\|^2$$

Step 2: set centroids $\boldsymbol{\mu}$ given $\mathbf{z}$

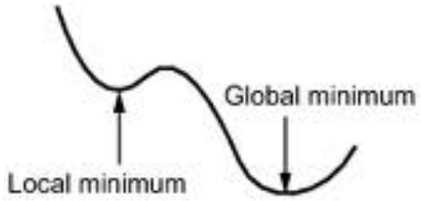
For each cluster $k = 1, \ldots, K$:

$$\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i:z_i=k} \phi(x_i)$$

- Now we can state the K-means algorithm formally. We start by initializing all the centroids randomly. Then, we iteratively alternate back and forth between steps 1 and 2, optimizing $\mathbf{z}$ given $\boldsymbol{\mu}$ and vice-versa.
- **Step 1** of K-means fixes the centroids $\boldsymbol{\mu}$. Then we can optimize the K-means objective with respect to $\mathbf{z}$ alone quite easily. It is easy to show that the best label for z_i is the cluster k that minimizes the distance to the centroid μ_k (which is fixed).
- **Step 2** turns things around and fixes the assignments $\mathbf{z}$. We can again look at the K-means objective function and optimize it with respect to the centroids $\boldsymbol{\mu}$. The best μ_k is to place the centroid at the average of all the points assigned to cluster k.

Local minima

K-means is guaranteed to converge to a local minimum, but is not guaranteed to find the global minimum.



[demo: getting stuck in local optima, seed $= 100$]

Solutions:

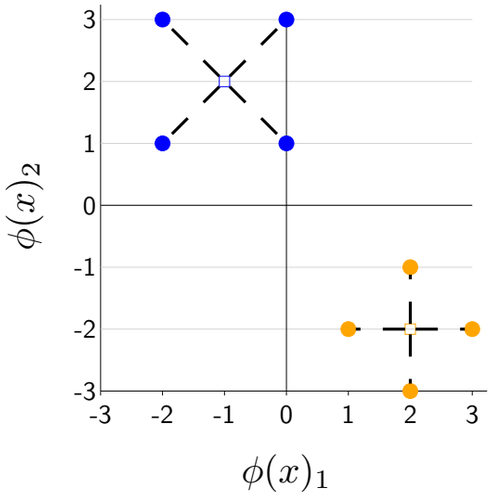
- Run multiple times from different random initializations
- Initialize with a heuristic (K-means++)

- K-means is guaranteed to decrease the loss function each iteration and will converge to a local minimum, but it is not guaranteed to find the global minimum, so one must exercise caution when applying K-means.
- Advanced: One solution is to simply run K-means several times from multiple random initializations and then choose the solution that has the lowest loss.
- Advanced: Or we could try to be smarter in how we initialize K-means. K-means++ is an initialization scheme which places centroids on training points so that these centroids tend to be distant from one another.

Summary

Clustering: discover structure in unlabeled data

K-means objective:



K-means algorithm:

assignments $\mathbf{z}$

centroids $\boldsymbol{\mu}$

Unsupervised learning use cases:

- Data exploration and discovery
- Providing representations to downstream supervised learning

- In summary, K-means is a simple and widely-used method for discovering cluster structure in data.
- Note that K-means can mean two things: the objective and the algorithm.
- Given points we define the K-means objective as the sum of the squared differences between a point and its assigned centroid.
- We also defined the K-means algorithm, which performs alternating optimization on the K-means objective.
- Finally, clustering is just one instance of unsupervised learning, which seeks to learn models from the wealth of unlabeled data alone. Unsupervised learning can be used in two ways: exploring a dataset which has not been labeled (let the data speak), and learning representations (discrete clusters or continuous embeddings) useful for downstream supervised applications.

Roadmap

Backpropagation

K-means

Generalization

Best practices

Summary of Machine Learning

- In this module, I will talk about the generalization of machine learning algorithms.

Minimizing training loss

Hypothesis class:

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

Training objective (loss function):

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

Optimization algorithm:

stochastic gradient descent

Is the training loss a good objective to optimize?

- Recall that our machine learning framework consists of specifying the hypothesis class, loss function, and the optimization algorithm.
- The hypothesis class could be linear predictors or neural networks. The loss function could be the hinge loss or the squared loss, which is averaged to produce the training loss.
- The default optimization algorithm is (stochastic) gradient descent.
- But let's be a bit more critical about the training loss. Is the training loss the right thing to optimize?

A strawman algorithm

Algorithm: rote learning

Training: just store $\mathcal{D}_{\text{train}}$.

Predictor $f(x)$:

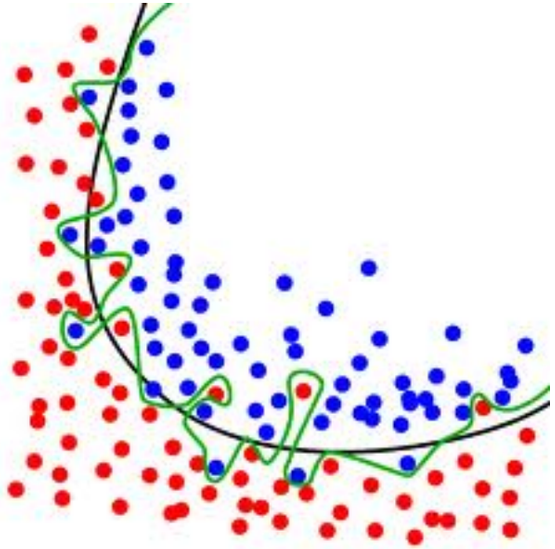
If $(x, y) \in \mathcal{D}_{\text{train}}$: return y.

Else: **segfault**.

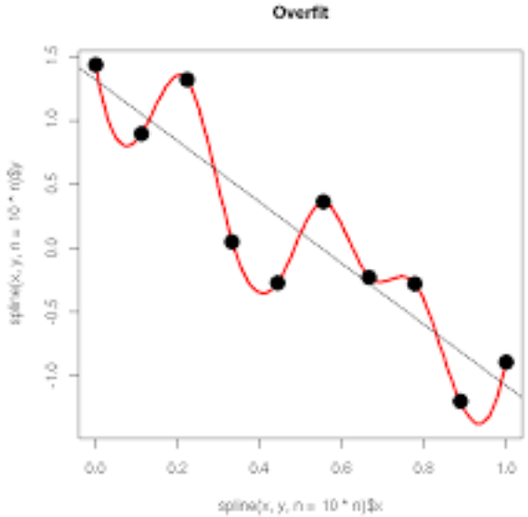
Minimizes the objective perfectly (zero), but clearly bad...

- Here is a strategy to consider: the rote learning algorithm, which just memorizes the training data and crashes otherwise.
- The rote learning algorithm does a perfect job of minimizing the training loss.
- But it's clearly a bad idea: It **overfits** to the training data and doesn't **generalize** to unseen examples.
- So clearly machine learning can't be about just minimizing the training loss.

Overfitting pictures



Classification



Regression

- This is an extreme example of **overfitting**, which is when a learning algorithm outputs a predictor that does well on the training data but not well on new examples.
- Here are two pictures that illustrate what overfitting looks like for binary classification and regression.
- On the left, we see that the green decision boundary gets zero training loss by separating all the blue points from the red ones. However, the smoother and simpler black curve is intuitively more likely to be the better classifier.
- On the right, we see that the predictor that goes through all the points will get zero training loss, but intuitively, the black line is perhaps a better option.
- In both cases, what is happening is that by over-optimizing on the training set, we risk fitting **noise** in the data.

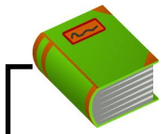
Evaluation

How good is the predictor f?

Key idea: the real learning objective

Our goal is to minimize **error on unseen future examples**.

Don't have unseen examples; next best thing:



Definition: test set

Test set $\mathcal{D}_{\text{test}}$ contains examples not used for training.

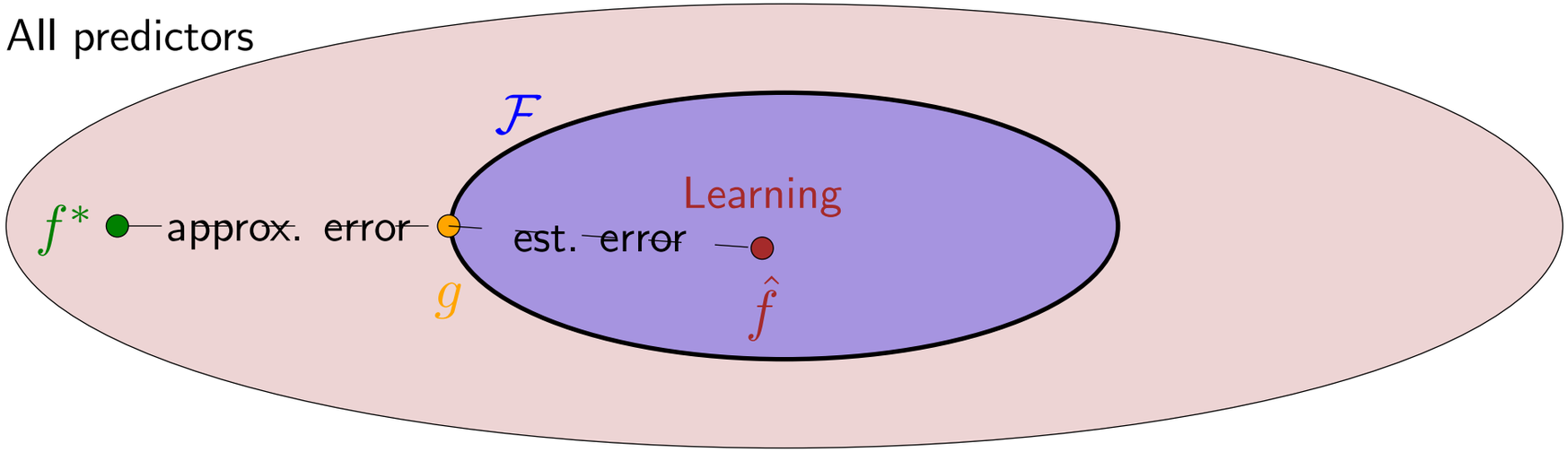
- So what is the true objective then? Taking a step back, machine learning is just a means to an end. What we're really doing is building a predictor to be deployed in the real world, and we just happen to be using machine learning. What we really care about is how accurate that predictor is on those **unseen future** inputs.
- Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that's unseen. We definitely should not tune our predictor based on the test set, because we wouldn't be able to do that on future examples.
- Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well your predictor is actually doing.

Generalization

When will a learning algorithm **generalize** well?

- So far, we have an intuitive feel for what overfitting is. How do we make this precise? In particular, when does a learning algorithm generalize from the training set to the test set?

Approximation and estimation error

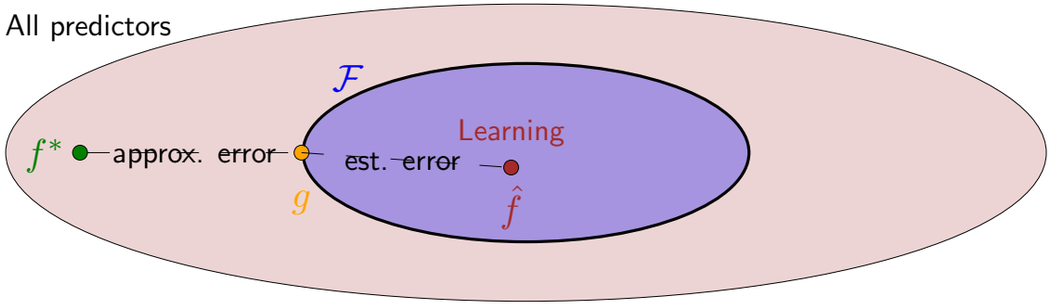


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\text{Err}(\hat{f}) - \text{Err}(f^*) = \underbrace{\text{Err}(\hat{f}) - \text{Err}(g)}_{\text{estimation}} + \underbrace{\text{Err}(g) - \text{Err}(f^*)}_{\text{approximation}}$$

- Here's a cartoon that can help you understand the balance between fitting and generalization. Out there somewhere, there is a magical predictor f^* that classifies everything perfectly. This predictor is unattainable; all we can hope to do is to use a combination of our domain knowledge and data to approximate that. The question is: how far are we away from f^*?
- Recall that our learning framework consists of (i) choosing a hypothesis class $\mathcal{F}$ (e.g., by defining the feature extractor) and then (ii) choosing a particular predictor $\hat{f}$ from $\mathcal{F}$.
- **Approximation error** is how far the entire hypothesis class is from the target predictor f^*. Larger hypothesis classes have lower approximation error. Let $g \in \mathcal{F}$ be the best predictor in the hypothesis class in the sense of minimizing test error $g = \arg\min_{f \in \mathcal{F}} \text{Err}(f)$. Here, distance is just the differences in test error: $\text{Err}(g) - \text{Err}(f^*)$.
- **Estimation error** is how good the predictor $\hat{f}$ returned by the learning algorithm is with respect to the best in the hypothesis class: $\text{Err}(\hat{f}) - \text{Err}(g)$. Larger hypothesis classes have higher estimation error because it's harder to find a good predictor based on limited data.
- We'd like both approximation and estimation errors to be small, but there's a tradeoff here.

Effect of hypothesis class size



As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

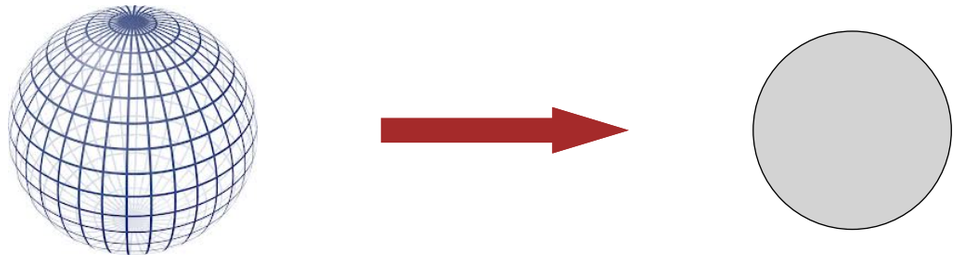
How do we control the hypothesis class size?

- The approximation error decreases monotonically as the hypothesis class size increases for a simple reason: you're taking a minimum over a larger set.
- The estimation error increases monotonically as the hypothesis class size increases for a deeper reason involving statistical learning theory (explained in CS229T).

Strategy 1: dimensionality

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

Reduce the dimensionality d (number of features):



- Let's focus our attention to linear predictors. For each weight vector $\mathbf{w}$, we have a predictor $f_{\mathbf{w}}$ (for classification, $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$). So the hypothesis class $\mathcal{F} = \{f_{\mathbf{w}}\}$ is all the predictors as $\mathbf{w}$ ranges. By controlling the number of possible values of $\mathbf{w}$ that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.
- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

Controlling the dimensionality

Manual feature (template) selection:

- Add feature templates if they help
- Remove feature templates if they don't help

Automatic feature selection (beyond the scope of this class):

- Forward selection
- Boosting
- L_1 regularization

It's the number of features that matters

- Mathematically, you can think about removing a feature $\phi(x)_{37}$ as simply only allowing its corresponding weight to be zero ($w_{37} = 0$).
- Operationally, if you have a few feature templates, then it's probably easier to just manually include or exclude them — this will give you more intuition.
- If you have a lot of individual features, you can apply more automatic methods for selecting features, but these are beyond the scope of this class.
- An important point is that it's the number of features that matters, not the number of feature templates. (Can you define one feature template that results in severe overfitting?) Nor is it the amount of code that you have to write to generate a particular feature.

Strategy 2: norm

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

Reduce the norm (length) $\|\mathbf{w}\|$:

- A related way to keep the weights small is called **regularization**, which involves adding an additional term to the objective function which penalizes the norm (length) of $\mathbf{w}$. This is probably the most common way to control the norm.

Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \lambda \mathbf{w})$$

Same as gradient descent, except shrink the weights towards zero by λ.

- This form of regularization is also known as L_2 regularization, or weight decay in deep learning literature.
- We can use gradient descent on this regularized objective, and this simply leads to an algorithm which subtracts a scaled down version of $\mathbf{w}$ in each iteration. This has the effect of keeping $\mathbf{w}$ closer to the origin than it otherwise would be.
- Note: Support Vector Machines are exactly hinge loss + L_2 regularization.

Controlling the norm: early stopping

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$

Idea: simply make T smaller

Intuition: if have fewer updates, then $\|\mathbf{w}\|$ can't get too big.

Lesson: try to minimize the training error, but don't try too hard.

- A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.
- Each time we update the weights, $\mathbf{w}$ has the potential of getting larger, so by running gradient descent a fewer number of iterations, we are implicitly ensuring that $\mathbf{w}$ stays small.
- Though early stopping seems hacky, there is actually some theory behind it. And one paradoxical note is that we can sometimes get better solutions by performing less computation.

Summary

Not the real objective: training loss

Real objective: loss on unseen future examples

Semi-real objective: test loss

Key idea: keep it simple

Try to minimize training error, but keep the hypothesis class small.

- In summary, we started by noting that the training loss is not the objective. Instead it is minimizing unseen future examples, which is approximated by the test set provided you are careful.
- We've seen several ways to control the size of the hypothesis class (and thus reducing the estimation error) based on either reducing the dimensionality or reducing the norm.
- It is important to note that what matters is the **size** of the hypothesis class, not how "complex" the predictors in the hypothesis class look. To put it another way, using complex features backed by 1000 lines of code doesn't hurt you if there are only 5 of them.
- So far, we've talked about the various knobs that we can turn to control the size of the hypothesis class, but how much do we turn each knob?

Roadmap

Backpropagation

K-means

Generalization

Best practices

Summary of Machine Learning

- We've spent a lot of talking about the formal principles of machine learning.
- In this module, I will discuss some of the more empirical aspects you encounter in practice.

Choose your own adventure

Hypothesis class:

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

Feature extractor ϕ: linear, quadratic

Architecture: number of layers, number of hidden units

Training objective:

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

Loss function: hinge, logistic

Regularization: none, L2

Optimization algorithm:

Algorithm: stochastic gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

  For $(x, y) \in \mathcal{D}_{\text{train}}$:

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

Number of epochs

Step size: constant, decreasing, adaptive

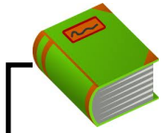
Initialization: amount of noise, pre-training

Batch size

Dropout

- Recall that there are three design decisions for setting up a machine learning algorithm: the hypothesis class, the training objective, and the optimization algorithm.
- For the hypothesis class, there are two knobs you can turn. The first is the feature extractor ϕ (linear features, quadratic features, indicator features on regions, etc. The second is the architecture of the predictor: linear (one layer) or neural network with layers, and in the case of neural networks, how many hidden units (k) do we have.
- The second design decision is to specify the training objective, which we do by specifying the loss function depending how we want the predictor to fit our data, and also whether we want to regularize the weights to guard against overfitting.
- The final design decision is how to optimize the predictor. Even the basic stochastic gradient descent algorithm has at least two knobs: how long to train (number of epochs) and how aggressively to update (the step size). On top of that are many enhancements and tweaks common to training deep neural networks: changing the step size over time, perhaps adaptively, how we initialize the weights, whether we update on batches (say of 16 examples) instead of 1, and whether we apply dropout to guard against overfitting.
- So it is really a choose your own machine learning adventure. Sometimes decisions can be made via prior knowledge and are thoughtful (e.g., features that capture periodic trends). But in many (even most) cases, we don't really know what the proper values should be. Instead, we want a way to have these just set automatically.

Hyperparameters



Definition: hyperparameters

Design decisions (hypothesis class, training objective, optimization algorithm) that need to be made before running the learning algorithm.

How do we choose hyperparameters?

Choose hyperparameters to minimize $\mathcal{D}_{\text{train}}$ error?

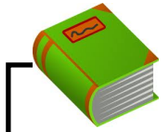
No - optimum would be to include all features, no regularization, train forever

Choose hyperparameters to minimize $\mathcal{D}_{\text{test}}$ error?

No - choosing based on $\mathcal{D}_{\text{test}}$ makes it an unreliable estimate of error!

- Each of these many design decisions is a **hyperparameter**.
- We could choose the hyperparameters to minimize the training loss. However, this would lead to a degenerate solution. For example, by adding additional features, we can always decrease the training loss, so we would just end up adding all the features in the world, leading to a model that wouldn't generalize. We would turn off all regularization, because that just gets in the way of minimizing the training loss.
- What if we instead chose hyperparameters to minimize the test loss. This might lead to good hyperparameters, but is problematic because you then lose the ability to measure how well you're doing. Recall that the test set is supposed to be a surrogate for unseen examples, and the more you optimize over them, the less unseen they become.

Validation set



Definition: validation set

A **validation set** is taken out of the training set and used to optimize hyperparameters.

For each setting of hyperparameters, train on $\mathcal{D}_{\text{train}} \setminus \mathcal{D}_{\text{val}}$, evaluate on $\mathcal{D}_{\text{val}}$

- The solution is to invent something that looks like a test set. There's no other data lying around, so we'll have to steal it from the training set. The resulting set is called the **validation set**.
- The size of the validation set should be large enough to give you a reliable estimate, but you don't want to take away too many examples from the training set.
- With this validation set, now we can simply try out a bunch of different hyperparameters and choose the setting that yields the lowest error on the validation set. Which hyperparameter values should we try? Generally, you should start by getting the right order of magnitude (e.g., $\lambda = 0.0001, 0.001, 0.01, 0.1, 1, 10$) and then refining if necessary.
- In K-fold **cross-validation**, you divide the training set into K parts. Repeat K times: train on $K-1$ of the parts and use the other part as a validation set. You then get K validation errors, from which you can compute and report both the mean and the variance, which gives you more reliable information.

Model development strategy

Algorithm: Model development strategy

- Split data into train, validation, test
- Look at data to get intuition
- Repeat:
  - Implement model/feature, adjust hyperparameters
  - Run learning algorithm
  - Sanity check train and validation error rates
  - Look at weights and prediction errors
- Evaluate on test set to get final error rates

- This slide represents the most important yet most overlooked part of machine learning: how to actually apply it in practice.
- We have so far talked about the mathematical foundation of machine learning (loss functions and optimization), and discussed some of the conceptual issues surrounding overfitting, generalization, and the size of hypothesis classes. But what actually takes most of your time is not writing new algorithms, but going through a **development cycle**, where you iteratively improve your system.
- The key is to stay connected with the data and the model, and have intuition about what's going on. Make sure to empirically examine the data before proceeding to the actual machine learning. It is imperative to understand the nature of your data in order to understand the nature of your problem.
- First, maintain data hygiene. Hold out a test set from your data that you don't look at until you're done. Start by looking at the (training or validation) data to get intuition. You can start to brainstorm what features / predictors you will need. You can compute some basic statistics.
- Then you enter a loop: implement a new model architecture or feature template. There are three things to look at: error rates, weights, and predictions. First, sanity check the error rates and weights to make sure you don't have an obvious bug. Then do an **error analysis** to see which examples your predictor is actually getting wrong. The art of practical machine learning is turning these observations into new features.
- Finally, run your system once on the test set and report the number you get. If your test error is much higher than your validation error, then you probably did too much tweaking and were **overfitting** (at a meta-level) the validation set.

Tips

Start simple:

- Run on small subsets of your data or synthetic data
- Start with a simple baseline model
- Sanity check: can you overfit 5 examples

Log everything:

- Track training loss and validation loss over time
- Record hyperparameters, statistics of data, model, and predictions
- Organize experiments (each run goes in a separate folder)

Report your results:

- Run each experiment multiple times with different random seeds
- Compute multiple metrics (e.g., error rates for minority groups)

- There is more to be said about the practice of machine learning. Here are some pieces of advice. Note that many related to simply good software engineering practices.
- First, don't start out by coding up a large complex model and try running it on a million examples. Start simple, both with the data (small number of examples) and the model (e.g., linear classifier). Sanity check that things are working first before increasing the complexity. This will help you debug in a regime where things are more interpretable and also things run faster. One sanity check is to train a sufficiently expressive model on a very few examples and see if the model can overfit the examples (get zero training error). This does not produce a useful model, but is a diagnostic to see if the optimization is working. If you can't overfit on 5 examples, then you have a problem: maybe the hypothesis class is too small, the data is too noisy, or the optimization isn't working.
- Second, log everything so you can diagnose problems. Monitor the losses over epochs. It is also important to track the training loss so that if you get bad results, you can find out if it is due to bad optimization or overfitting. Record all the hyperparameters, so that you have a full record of how to reproduce the results.
- Third, when you report your results, you should be able to run an experiment multiple times with different randomness to see how stable the results are. Report error bars. And finally, if it makes sense for your application to report more than just a single test accuracy. Report the errors for minority groups and add if your model is treating every group fairly.

Summary

Don't look at the test set!

Understand the data!

Start simple!

Practice!

- To summarize, we've talked about the practice of machine learning.
- First, make sure you follow good data hygiene, separating out the test set and don't look at it.
- But you should look at the training or validation set to get intuition about your data before you start.
- Then, start simple and make sure you understand how things are working.
- Beyond that, there are a lot of design decisions to be made (hyperparameters). So the most important thing is to practice, so that you can start developing more intuition, and developing a set of best practices that works for you.

Roadmap

Backpropagation

K-means

Generalization

Best practices

Summary of Machine Learning

Machine Learning Summary

- Feature extraction (think hypothesis classes) [modeling]
- Prediction (linear, neural network, k-means) [modeling]
- Loss functions (evaluate errors) [modeling]
- Optimization (stochastic gradient, alternating minimization) [learning]
- Generalization (think development cycle) [modeling]
- We are not covering some other important aspects, e.g., fairness, privacy, interpretability

- This concludes our tour of the foundations of machine learning, although machine learning will come up again later in the course. You should have gotten more than just a few isolated equations and algorithms. It is really important to think about the overarching principles in a modular way.
- First, feature extraction is where you put your domain knowledge into. In designing features, it's useful to think in terms of the induced **hypothesis classes** — what kind of functions can your learning algorithm potentially learn?
- These features then drive **prediction**: either linearly or through a neural network. We can even think of k-means as trying to predict the data points using the centroids.
- **Loss functions** connect predictions with the actual training examples.
- Note that all of the design decisions up to this point are about modeling. Algorithms are very important, but only come in once we have the right **optimization problem** to solve.
- Finally, machine learning requires a leap of faith. How does optimizing anything at training time help you **generalize** to new unseen examples at test time? Learning can only work when there's a common core that cuts past all the idiosyncrasies of the examples. This is exactly what features are meant to capture.
- Note that our lectures are a limited tour of machine learning, and do not cover some important aspects e.g., fairness, privacy, interpretability in machine learning

Machine learning

Key idea: learning

Programs should improve with experience.

So far: reflex-based models

Next time: state-based models

- If we generalize for a moment, machine learning is really about programs that can improve with experience.
- So far, we have only focused on reflex-based models where the program only outputs a yes/no or a number, and the experience is examples of input-output pairs.
- Next time, we will start looking at models which can perform higher-level reasoning, but machine learning will remain our companion for the remainder of the class.