

CSC421/2516 Lecture 3: AutoDiff, Distributed Representations & Optimization

Jimmy Ba and Bo Wang

Autodiff

- Lecture 2 covered the algebraic view of backprop.
- Here, we'll see how to implement an automatic differentiation library:
 - build the computation graph
 - vector-Jacobian products (VJP) for primitive ops
 - the backwards pass
- We'll use Autograd, a lightweight autodiff tool, as an example. The implementations of PyTorch, TensorFlow, Jax, etc. are very similar.
 - You will probably never have to implement autodiff yourself but it is good to know its inner workings.

Confusing Terminology

- **Automatic differentiation (autodiff)** refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
- **Backpropagation** is the special case of autodiff applied to neural nets
 - But in machine learning, we often use backprop synonymously with autodiff
- **Autograd** is the name of a particular autodiff library we will cover in this lecture. There are many others, e.g. PyTorch, TensorFlow.

What is Autodiff

- An autodiff system will convert the program into a sequence of **primitive operations (ops)** which have specified routines for computing derivatives.
- In this representation, backprop can be done in a completely mechanical way.

Sequence of primitive operations:

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

What is Autodiff

```
import autograd.numpy as np ← very sneaky!
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

... (load the data) ...

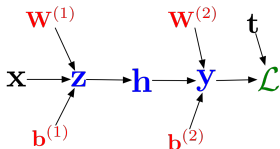
```
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print "Trained loss:", training_loss(weights)
```

Autograd constructs a function for computing derivatives

Building the Computation Graph

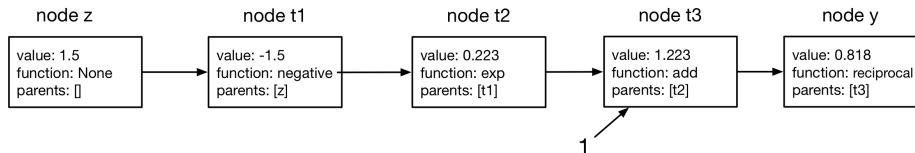


- Most autodiff systems, including Autograd, explicitly construct the computation graph.
 - Some frameworks like TensorFlow provide mini-languages for building computation graphs directly. Disadvantage: need to learn a totally new API.
 - Autograd instead builds them by **tracing** the forward pass computation, allowing for an interface nearly indistinguishable from NumPy.
- The **Node** class (defined in `tracer.py`) represents a node of the computation graph. It has attributes:
 - `value`, the actual value computed on a particular set of inputs
 - `fun`, the primitive operation defining the node
 - `args` and `kwargs`, the arguments the op was called with
 - `parents`, the parent Nodes

Building the Computation Graph

- Autograd's fake NumPy module provides primitive ops which look and feel like NumPy functions, but secretly build the computation graph.
- Example:

```
def logistic(z):  
    return 1. / (1. + np.exp(-z))  
  
# that is equivalent to:  
def logistic2(z):  
    return np.reciprocal(np.add(1, np.exp(np.negative(z))))  
  
z = 1.5  
y = logistic(z)
```



Vector-Jacobian Products

- For each primitive operation, we must specify VJPs for *each* of its arguments. Consider $y = \exp(x)$.
- This is a function which takes in the output gradient (i.e. \bar{y}), the answer (y), and the arguments (x), and returns the input gradient (\bar{x})
- `defvjp` (defined in `core.py`) is a convenience routine for registering VJPs. It just adds them to a dict.
- Examples from `numpy/numpy_vjps.py`

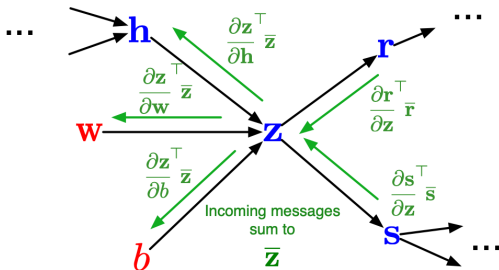
```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp,      lambda g, ans, x: ans * g)
defvjp(log,     lambda g, ans, x: g / x)

defvjp(add,      lambda g, ans, x, y : g,
              lambda g, ans, x, y : g)
defvjp(multiply, lambda g, ans, x, y : y * g,
              lambda g, ans, x, y : x * g)
defvjp(subtract, lambda g, ans, x, y : g,
              lambda g, ans, x, y : -g)
```


Backprop as Message Passing

Backprop as message passing:

- Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents.
- Each of these messages is a VJP.
- This formulation provides modularity: each node needs to know how to compute its outgoing messages, i.e. the VJPs corresponding to each of its parents (arguments to the function).
- The implementation of z doesn't need to know where \bar{z} came from.



Backward Pass

- The backwards pass is defined in `core.py`.
- The argument `g` is the error signal for the end node; for us this is always $\bar{\mathcal{L}} = 1$.

```
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad

def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```

Put Everything Together

- `grad` (in `differential_operators.py`) is just a wrapper around `make_vjp` (in `core.py`) which builds the computation graph and feeds it to `backward_pass`.
- `grad` itself is viewed as a VJP, if we treat $\bar{\mathcal{L}}$ as the 1×1 matrix with entry 1.

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial w} \bar{\mathcal{L}}$$

```
def make_vjp(fun, x):
    """Trace the computation to build the computation graph, and return
    a function which implements the backward pass."""
    start_node = Node.new_root()
    end_value, end_node = trace(start_node, fun, x)
    def vjp(g):
        return backward_pass(g, end_node)
    return vjp, end_value

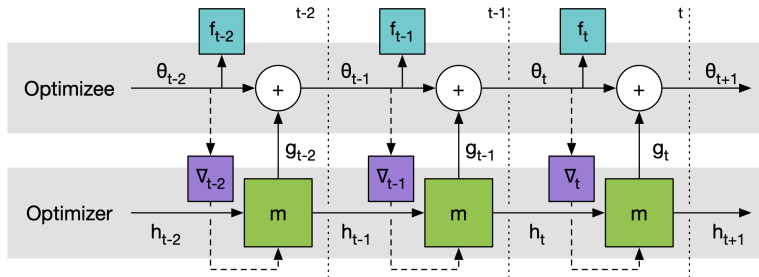
def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
```

Recap

- We saw three main parts to the code:
 - tracing the forward pass to build the computation graph
 - vector-Jacobian products for primitive ops
 - the backwards pass
- Building the computation graph requires fancy NumPy gymnastics, but other two items are basically what we have in the last two slides.
- You're encouraged to read the full code (< 200 lines!) at:

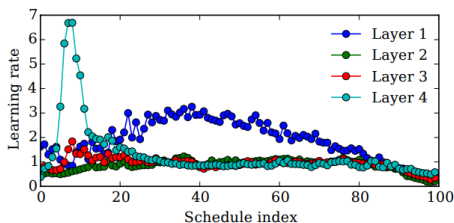
<https://github.com/mattjj/autodidact/tree/master/autograd>

Autodiff Application: Learning to learning by gradient descent by gradient descent

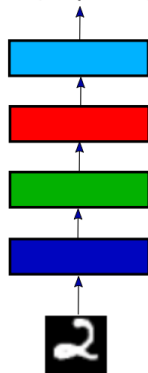


<https://arxiv.org/pdf/1606.04474.pdf>

Autodiff Application: Gradient-Based Hyperparameter Optimization



$P(\text{digit} \mid \text{image})$



<https://arxiv.org/abs/1502.03492>

After the break: **Distributed Representations**

Distributed Representations

- Let's now take a break from backpropagation and see a real example of a neural net to learn feature representations of words.
 - We'll see a lot more neural net architectures later in the course.
- We'll also introduce the models used in Programming Assignment 1.

Review: Probability and Bayes' Rule

Suppose we want to build a speech recognition system.

We'd like to be able to infer a likely sentence s given the observed speech signal a . The **generative** approach is to build two components:

- An **observation model**, represented as $p(a | s)$, which tells us how likely the sentence s is to lead to the acoustic signal a .
- A **prior**, represented as $p(s)$, which tells us how likely a given sentence s is. E.g., it should know that “recognize speech” is more likely than “wreck a nice beach.”

Given these components, we can use **Bayes' Rule** to infer a **posterior distribution** over sentences given the speech signal:

$$p(s | a) = \frac{p(s)p(a | s)}{\sum_{s'} p(s')p(a | s')}.$$

Language Modeling

From here on, we will focus on learning a good distribution $p(s)$ of sentences. This problem is known as **language modeling**.

Assume we have a corpus of sentences $s^{(1)}, \dots, s^{(N)}$. The **maximum likelihood** criterion says we want our model to maximize the probability our model assigns to the observed sentences. We assume the sentences are independent, so that their probabilities multiply.

$$\max \prod_{i=1}^N p(s^{(i)}).$$

Language Modeling

In maximum likelihood training, we want to maximize $\prod_{i=1}^N p(s^{(i)})$.

The probability of generating the whole training corpus is vanishingly small — like monkeys typing all of Shakespeare.

- The **log probability** is something we can work with more easily. It also conveniently decomposes as a sum:

$$\log \prod_{i=1}^N p(s^{(i)}) = \sum_{i=1}^N \log p(s^{(i)}).$$

- This is equivalent to the **cross-entropy loss**.

Language Modeling

- Probability of a sentence? What does that even mean?
 - A sentence is a sequence of words w_1, w_2, \dots, w_T . Using the **chain rule of conditional probability**, we can decompose the probability as
$$p(s) = p(w_1, \dots, w_T) = p(w_1)p(w_2 | w_1) \cdots p(w_T | w_1, \dots, w_{T-1}).$$
 - Therefore, the language modeling problem is equivalent to being able to predict the next word!
- We typically make a **Markov assumption**, i.e. that the distribution over the next word only depends on the preceding few words. I.e., if we use a context of length 3,

$$p(w_t | w_1, \dots, w_{t-1}) = p(w_t | w_{t-3}, w_{t-2}, w_{t-1}).$$

- Such a model is called **memoryless**.
- Now it's basically a supervised prediction problem. We need to predict the conditional distribution of each word given the previous K .
- When we decompose it into separate prediction problems this way, it's called an **autoregressive model**.

N-Gram Language Models

- One sort of Markov model we can learn uses a **conditional probability table**, i.e.

	cat	and	city	...
the fat	0.21	0.003	0.01	
four score	0.0001	0.55	0.0001	...
New York	0.002	0.0001	0.48	
⋮		⋮		

- Maybe the simplest way to estimate the probabilities is from the **empirical distribution**:

$$p(w_3 = \text{cat} \mid w_1 = \text{the}, w_2 = \text{fat}) = \frac{p(w_1 = \text{the}, w_2 = \text{fat}, w_3 = \text{cat})}{p(w_1 = \text{the}, w_2 = \text{fat})}$$
$$\approx \frac{\text{count}(\text{the fat cat})}{\text{count}(\text{the fat})}$$

- The phrases we're counting are called **n-grams** (where n is the length), so this is an **n-gram language model**.
 - So, the above example is considered a 3-gram model.

N-Gram Language Models

- Problems with n-gram language models
 - The number of entries in the conditional probability table is exponential in the context length.
 - **Data sparsity**: most n-grams never appear in the corpus, even if they are possible.
- Traditional ways to deal with data sparsity

N-Gram Language Models

- Problems with n-gram language models
 - The number of entries in the conditional probability table is exponential in the context length.
 - **Data sparsity**: most n-grams never appear in the corpus, even if they are possible.
- Traditional ways to deal with data sparsity
 - Use a short context (but this means the model is less powerful)
 - Smooth the probabilities, e.g. by adding imaginary counts
 - Make predictions using an ensemble of n-gram models with different n

Distributed Representations

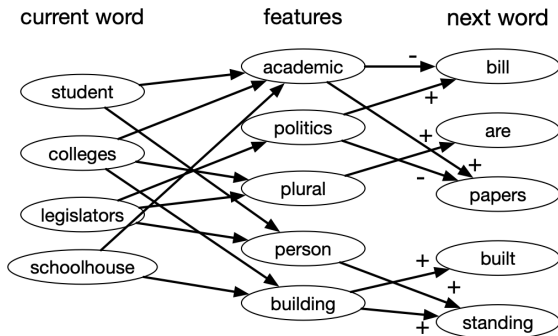
- Conditional probability tables are a kind of **localist representation**: all the information about a particular word is stored in one place, i.e. a column of the table.
- But different words are related, so we ought to be able to **share** information between them. For instance, consider this matrix of word attributes:

	academic	politics	plural	person	building
students	1	0	1	1	0
colleges	1	0	1	0	1
legislators	0	1	1	1	0
schoolhouse	1	0	0	0	1

- And this matrix of how each attribute influences the next word:

	bill	is	are	papers	built	standing
academic	-			+		
politics	+			-		
plural		-	+			
person						+
building					+	+

- Imagine these matrices are layers in an MLP. (One-hot representations of words, softmax over next word.)



- Here, the information about a given word is distributed throughout the representation. We call this a **distributed representation**.
- In general, when we train an MLP with backprop, the hidden units won't have intuitive meanings like in this cartoon. But this is a useful intuition pump for what MLPs can represent.

Distributed Representations

- We would like to be able to share information between related words.
E.g., suppose we've seen the sentence
The cat got squashed in the garden on Friday.
- This should help us predict the previously unseen words
The dog got flattened in the yard on Monday.
- An n-gram model can't generalize this way, but a distributed representation might let us do so.

Neural Language Model

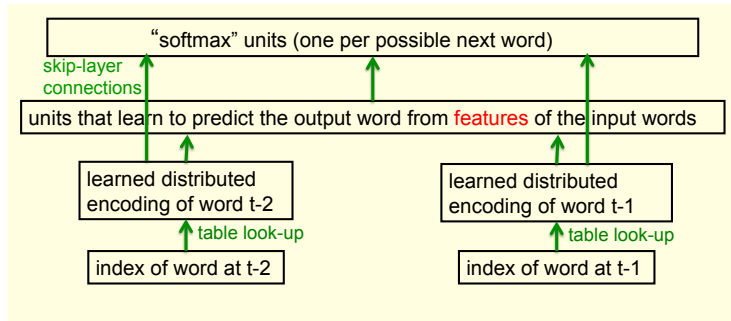
- Predicting the distribution of the next word given the previous K is just a multiway classification problem.
- **Inputs:** previous K words
- **Target:** next word
- **Loss:** cross-entropy. Recall that this is equivalent to maximum likelihood:

$$\begin{aligned} -\log p(s) &= -\log \prod_{t=1}^T p(w_t | w_1, \dots, w_{t-1}) \\ &= -\sum_{t=1}^T \log p(w_t | w_1, \dots, w_{t-1}) \\ &= -\sum_{t=1}^T \sum_{v=1}^V t_{tv} \log y_{tv}, \end{aligned}$$

where t_{iv} is the one-hot encoding for the i th word and y_{iv} is the predicted probability for the i th word being index v .

Bengio's Neural Language Model

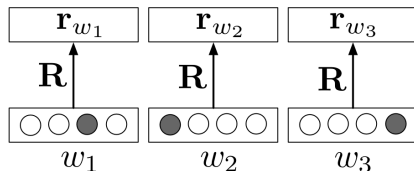
- Here is a classic neural probabilistic language model, or just neural language model:



<http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf>

Neural Language Model

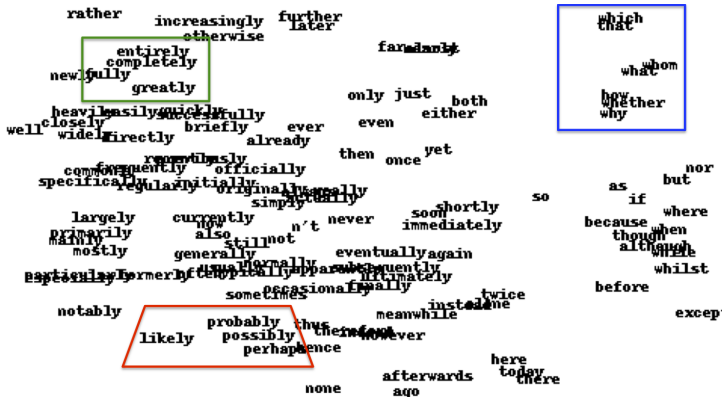
- If we use a 1-of-K encoding for the words, the first layer can be thought of as a linear layer with **tied weights**.



- The weight matrix basically acts like a lookup table. Each column is the **representation** of a word, also called an **embedding**, **feature vector**, or **encoding**.
 - “Embedding” emphasizes that it’s a location in a high-dimensional space; words that are closer together are more semantically similar
 - “Feature vector” emphasizes that it’s a vector that can be used for making predictions, just like other feature mappings we’ve looked at (e.g. polynomials)

Neural Language Model

- What do these word embeddings look like?
- The following 2-D embeddings are done with an algorithm called tSNE which tries to make distances in the 2-D embedding match the original 30-D distances as closely as possible.



Neural Language Model



Neural Language Model

- Thinking about high-dimensional embeddings
 - Most vectors are nearly orthogonal (i.e. dot product is close to 0)
 - Most points are far away from each other
 - “In a 30-dimensional grocery store, anchovies can be next to fish and next to pizza toppings.” – Geoff Hinton
- The 2-D embeddings might be fairly misleading, since they can't preserve the distance relationships from a higher-dimensional embedding. (I.e., unrelated words might be close together in 2-D, but far apart in 30-D.)

GloVe: Neural Language Model as Matrix Factorization

- Fitting language models is really hard.
- Maybe this is overkill if all you want is word representations.
- Global Vector (GloVe) embeddings are a simpler and faster approach based on a matrix factorization similar to principal component analysis (PCA).

GloVe: Neural Language Model as Matrix Factorization

- **Distributional hypothesis** again: words with similar distributions have similar meanings (“judge a word by the company it keeps”)
- Consider a **co-occurrence matrix** X , which counts the number of times two words appear nearby (say, less than 5 positions apart)
- This is a $V \times V$ matrix, where V is the vocabulary size (very large)
- **Intuition pump:** suppose we fit a rank- K approximation $X \approx R\tilde{R}^T$, where R and \tilde{R} are $V \times K$ matrices.
 - Each row r_i of R is the K -dimensional representation of a word
 - Each entry is approximated as $x_{ij} \approx r_i^T \tilde{r}_j$
 - Hence, more similar words are more likely to co-occur
 - **Minimizing the squared Frobenius norm**
 $\|X - R\tilde{R}^T\|_F^2 = \sum_{i,j} (x_{ij} - r_i^T \tilde{r}_j)^2$ is basically PCA.

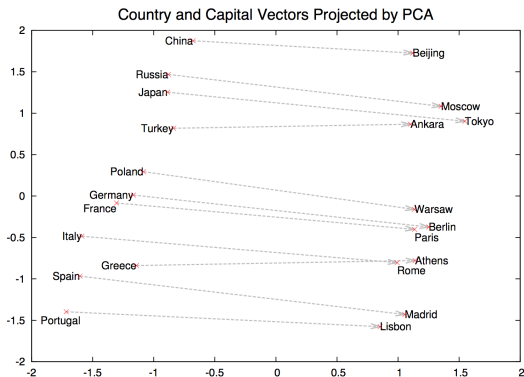
- **Problem 1:** X is extremely large, so fitting the above factorization using least squares is infeasible.
 - **Solution:** Reweight the entries so that only nonzero counts matter
- **Problem 2:** Word counts are a heavy-tailed distribution, so the most common words will dominate the cost function.
 - **Solution:** Approximate $\log x_{ij}$ instead of x_{ij} .
- **Global Vector (GloVe) embedding** cost function:

$$\mathcal{J}(R) = \sum_{i,j} f(x_{ij})(r_i^\top \tilde{r}_j + b_i + \tilde{b}_j - \log x_{ij})^2$$
$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{100}\right)^{3/4} & \text{if } x_{ij} < 100 \\ 1 & \text{if } x_{ij} \geq 100 \end{cases}$$

- b_i and \tilde{b}_j are bias parameters.
- We can avoid computing $\log 0$ since $f(0) = 0$.
- We only need to consider the nonzero entries of X . This gives a big computational savings since X is extremely sparse!

Word Analogies

- Here's a linear projection of word representations for cities and capitals into 2 dimensions.
- The mapping city \rightarrow capital corresponds roughly to a single direction in the vector space:



- Note: this figure actually comes from skip-grams, a predecessor to GloVe.

Word Analogies

- In other words,
 $\text{vector}(\text{Paris}) - \text{vector}(\text{France}) \approx \text{vector}(\text{London}) - \text{vector}(\text{England})$
- This means we can analogies by doing arithmetic on word vectors:
 - e.g. “Paris is to France as London is to _____”
 - Find the word whose vector is closest to
 $\text{vector}(\text{France}) - \text{vector}(\text{Paris}) + \text{vector}(\text{London})$
- Example analogies:

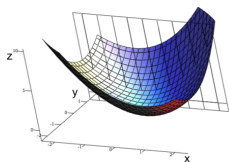
Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

After the break: **Optimization**

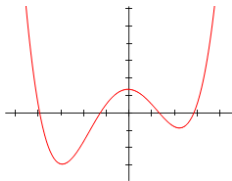
Optimization

- So far, we've talked a lot about computing gradients and different neural models.
- How do we actually train those models using gradients?
- There are various things that can go wrong in gradient descent, we will learn what to do about them, e.g.
 - How to tune the learning rates.
- For convenience in this part, let's group all the parameters (weights and biases) of our network into a single vector θ .

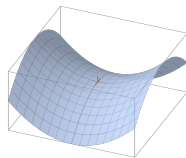
Features of the Optimization Landscape



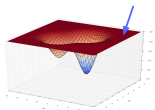
convex functions



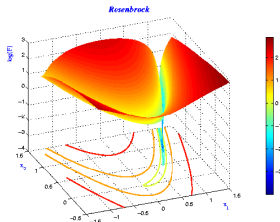
local minima



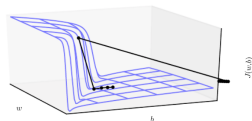
saddle points



plateaux



narrow ravines



cliffs (covered in a later lecture)

Review: Hessian Matrix

- The **Hessian matrix**, denoted H , or $\nabla^2 \mathcal{J}$ is the matrix of second derivatives:

$$H = \nabla^2 \mathcal{J} = \begin{pmatrix} \frac{\partial^2 \mathcal{J}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_D} \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_D^2} \end{pmatrix}$$

- It's a symmetric matrix because $\frac{\partial^2 \mathcal{J}}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 \mathcal{J}}{\partial \theta_j \partial \theta_i}$.

Review: Hessian Matrix

- Locally, a function can be approximated by its **second-order Taylor approximation** around a point θ_0 :

$$\mathcal{J}(\theta) \approx \mathcal{J}(\theta_0) + \nabla \mathcal{J}(\theta_0)^\top (\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \mathbf{H}(\theta_0)(\theta - \theta_0).$$

- A **critical point** is a point where the gradient is zero. In that case,

$$\mathcal{J}(\theta) \approx \mathcal{J}(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \mathbf{H}(\theta_0)(\theta - \theta_0).$$

Review: Hessian Matrix

- Why do we need Hessian: A lot of important features of the optimization landscape can be characterized by the eigenvalues of the Hessian H .
- Recall that a symmetric matrix (such as H) has only real eigenvalues, and there is an orthogonal basis of eigenvectors.
- This can be expressed in terms of the **spectral decomposition**:

$$H = Q\Lambda Q^T,$$

where Q is an orthogonal matrix (whose columns are the eigenvectors) and Λ is a diagonal matrix (whose diagonal entries are the eigenvalues).

Review: Hessian Matrix

- We often refer to H as the **curvature** of a function.
- Suppose you move along a line defined by $\boldsymbol{\theta} + t\mathbf{v}$ for some vector \mathbf{v} .
- Second-order Taylor approximation:

$$\mathcal{J}(\boldsymbol{\theta} + t\mathbf{v}) \approx \mathcal{J}(\boldsymbol{\theta}) + t\nabla\mathcal{J}(\boldsymbol{\theta})^\top\mathbf{v} + \frac{t^2}{2}\mathbf{v}^\top\mathbf{H}(\boldsymbol{\theta})\mathbf{v}$$

- Hence, in a direction where $\mathbf{v}^\top\mathbf{H}\mathbf{v} > 0$, the cost function curves upwards, i.e. has **positive curvature**. Where $\mathbf{v}^\top\mathbf{H}\mathbf{v} < 0$, it has **negative curvature**.

Review: Hessian Matrix

- A matrix A is **positive definite** if $v^T Av > 0$ for all $v \neq 0$. (I.e., it curves upwards in all directions.)
 - It is **positive semidefinite (PSD)** if $v^T Av \geq 0$ for all $v \neq 0$.
- Equivalently: a matrix is positive definite iff all its eigenvalues are positive. It is PSD iff all its eigenvalues are nonnegative. (Exercise: show this using the Spectral Decomposition.)
- For any critical point θ_* , if $H(\theta_*)$ exists and is positive definite, then θ_* is a local minimum (since all directions curve upwards).

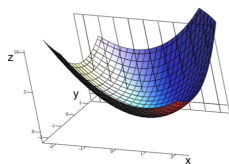
Review: Convex Functions

- Recall: a set \mathcal{S} is convex if for any $x_0, x_1 \in \mathcal{S}$,

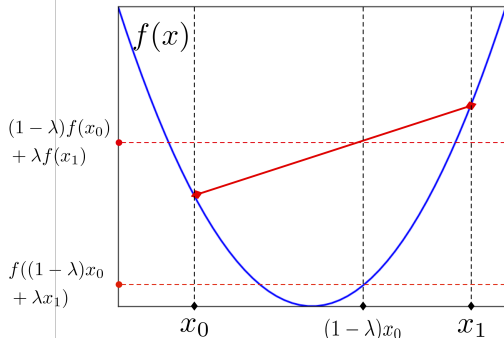
$$(1 - \lambda)x_0 + \lambda x_1 \in \mathcal{S} \quad \text{for } 0 \leq \lambda \leq 1.$$

- A function f is **convex** if for any x_0, x_1 ,

$$f((1 - \lambda)x_0 + \lambda x_1) \leq (1 - \lambda)f(x_0) + \lambda f(x_1)$$



- Equivalently, the set of points lying above the graph of f is convex.
- Intuitively: the function is bowl-shaped.

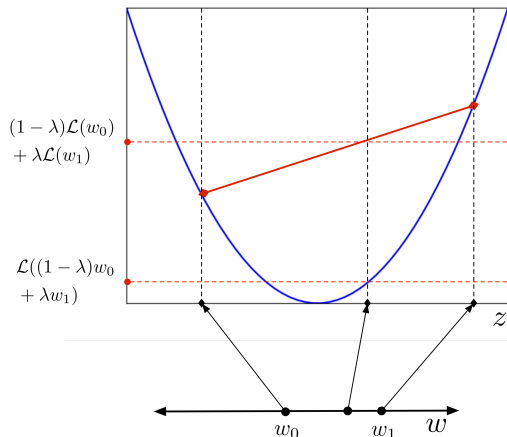


Review: Convex Functions

- If \mathcal{J} is **smooth** (more precisely, twice differentiable), there's an equivalent characterization in terms of H :
 - A smooth function is convex iff its Hessian is positive semidefinite everywhere.
- **Exercise:** show that squared error, logistic-cross-entropy, and softmax-cross-entropy losses are convex (as a function of the network outputs) by taking second derivatives.

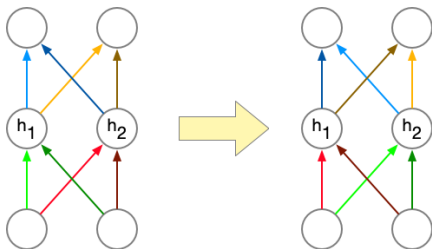
Review: Convex Functions

- For a linear model, $z = w^\top x + b$ is a linear function of w and b . If the loss function is convex as a function of z , then it is convex as a function of w and b .
- Hence, linear regression, logistic regression, and softmax regression are convex.



Local Minima

- If a function is convex, it has no **spurious local minima**, i.e. any local minimum is also a global minimum.
- This is very convenient for optimization since if we keep going downhill, we'll eventually reach a global minimum.
- Unfortunately, training a network with hidden units cannot be convex because of **permutation symmetries**.
 - I.e., we can re-order the hidden units in a way that preserves the function computed by the network.



Local Minima

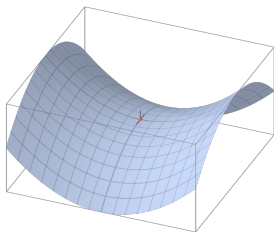
Special case: a univariate function is convex iff its second derivative is nonnegative everywhere.

- By definition, if a function \mathcal{J} is convex, then for any set of points $\theta_1, \dots, \theta_N$ in its domain,

$$\mathcal{J}(\lambda_1\theta_1 + \dots + \lambda_N\theta_N) \leq \lambda_1\mathcal{J}(\theta_1) + \dots + \lambda_N\mathcal{J}(\theta_N) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1.$$

- Because of permutation symmetry, there are $K!$ permutations of the hidden units in a given layer which all compute the same function.
- Suppose we average the parameters for all $K!$ permutations. Then we get a degenerate network where all the hidden units are identical.
- If the cost function were convex, this solution would have to be better than the original one, which is ridiculous!
- Hence, training multilayer neural nets is non-convex.

Saddle points



A **saddle point** is a point where:

- $\nabla \mathcal{J}(\boldsymbol{\theta}) = 0$
- $H(\boldsymbol{\theta})$ has some positive and some negative eigenvalues, i.e. some directions with positive curvature and some with negative curvature.

When would saddle points be a problem?

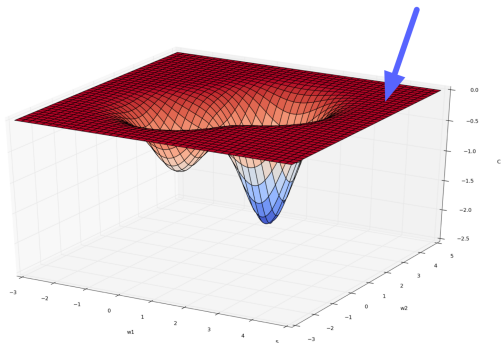
- If we're exactly on the saddle point, then we're stuck.
- If we're slightly to the side, then we can get unstuck.

Saddle points

- Suppose you have two hidden units with identical incoming and outgoing weights.
- After a gradient descent update, they will still have identical weights. By induction, they'll always remain identical.
- But if you perturbed them slightly, they can start to move apart.
- Important special case: don't initialize all your weights to zero!
 - Instead, **break the symmetry** by using small random values.

Plateaux

A flat region is called a **plateau**. (Plural: plateaux)



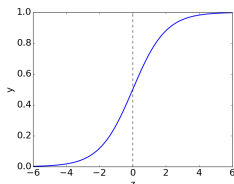
Examples of plateaux:

- 0-1 loss
- hard threshold activations
- logistic activations & least squares

Plateaux

- An important example of a plateau is a **saturated unit**. This is when it is in the flat region of its activation function. Recall the backprop equation for the weight derivative:

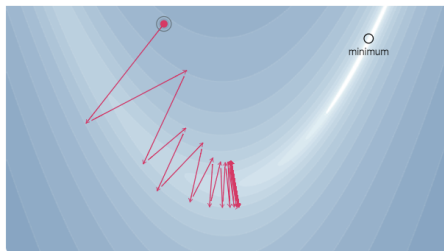
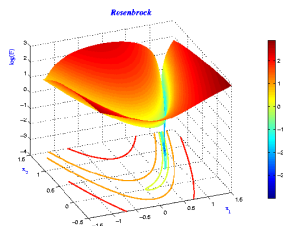
$$\bar{z}_i = \bar{h}_i \phi'(z)$$
$$\bar{w}_{ij} = \bar{z}_i x_j$$



- If $\phi'(z_i)$ is always close to zero, then the weights will get stuck.
- If there is a ReLU unit whose input z_i is always negative, the weight derivatives will be *exactly* 0. We call this a **dead unit**.

Ill-conditioned curvature

Long, narrow ravines:



- Suppose H has some large positive eigenvalues (i.e. high-curvature directions) and some eigenvalues close to 0 (i.e. low-curvature directions).
- Gradient descent bounces back and forth in high curvature directions and makes slow progress in low curvature directions.
 - To interpret this visually: the gradient is perpendicular to the contours.
- This is known as **ill-conditioned curvature**. It's very common in neural net training.

Ill-conditioned curvature: gradient descent dynamics

- To understand why ill-conditioned curvature is a problem, consider a convex quadratic objective

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{A} \boldsymbol{\theta},$$

where \mathbf{A} is PSD.

- Gradient descent update:

$$\begin{aligned} \boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k - \alpha \nabla \mathcal{J}(\boldsymbol{\theta}_k) \\ &= \boldsymbol{\theta}_k - \alpha \mathbf{A} \boldsymbol{\theta}_k \\ &= (\mathbf{I} - \alpha \mathbf{A}) \boldsymbol{\theta}_k \end{aligned}$$

- Solving the recurrence,

$$\boldsymbol{\theta}_k = (\mathbf{I} - \alpha \mathbf{A})^k \boldsymbol{\theta}_0$$

Ill-conditioned curvature: gradient descent dynamics

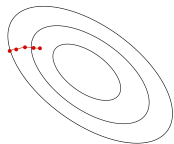
- We can analyze matrix powers such as $(I - \alpha A)^k \theta_0$ using the spectral decomposition.
- Let $A = Q\Lambda Q^\top$ be the spectral decomposition of A .

$$\begin{aligned}(I - \alpha A)^k \theta_0 &= (I - \alpha Q\Lambda Q^\top)^k \theta_0 \\ &= [Q(I - \alpha\Lambda)Q^\top]^k \theta_0 \\ &= Q(I - \alpha\Lambda)^k Q^\top \theta_0\end{aligned}$$

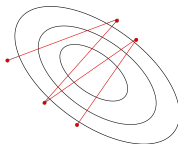
- Hence, in the Q basis, each coordinate gets multiplied by $(1 - \alpha\lambda_i)^k$, where the λ_i are the eigenvalues of A .
- Cases:
 - $0 < \alpha\lambda_i \leq 1$: decays to 0 at a rate that depends on $\alpha\lambda_i$
 - $1 < \alpha\lambda_i \leq 2$: oscillates
 - $\alpha\lambda_i > 2$: unstable (diverges)

Tuning Learning Rate

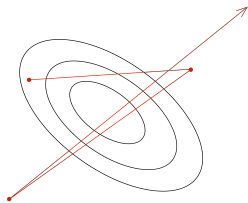
- How can spectral decomposition help?
- The learning rate α is a hyperparameter we need to tune. Here are the things that can go wrong:



α too small:
slow progress



α too large:
oscillations



α much too large:
instability (diverges)

Ill-conditioned curvature: gradient descent dynamics

- Just showed
 - $0 < \alpha\lambda_i \leq 1$: decays to 0 at a rate that depends on $\alpha\lambda_i$
 - $1 < \alpha\lambda_i \leq 2$: oscillates
 - $\alpha\lambda_i > 2$: unstable (diverges)
- **Ill-conditioned curvature bounds the maximum learning rate choice.** Need to set the learning rate $\alpha < 2/\lambda_{\max}$ to prevent instability, where λ_{\max} is the largest eigenvalue, i.e. maximum curvature.
- This bounds the rate of progress in another direction:

$$\alpha\lambda_i < \frac{2\lambda_i}{\lambda_{\max}}.$$

- The quantity $\lambda_{\max}/\lambda_{\min}$ is known as the **condition number** of A . Larger condition numbers imply slower convergence of gradient descent.

Ill-conditioned curvature: gradient descent dynamics

- The analysis we just did was for a quadratic toy problem

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{A} \boldsymbol{\theta}.$$

- It can be easily generalized to a quadratic not centered at zero, since the gradient descent dynamics are invariant to translation.

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2} \boldsymbol{\theta}^\top \mathbf{A} \boldsymbol{\theta} + \mathbf{b}^\top \boldsymbol{\theta} + c$$

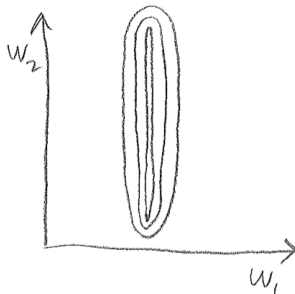
- Since a smooth cost function is well approximated by a convex quadratic (i.e. second-order Taylor approximation) in the vicinity of a (local) optimum, this analysis is a good description of the behavior of gradient descent near a (local) optimum.
- If the Hessian is ill-conditioned, then gradient descent makes slow progress towards the optimum.

Ill-conditioned curvature: normalization

- Suppose we have the following dataset for linear regression.

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
\vdots	\vdots	\vdots

$$\bar{w}_i = \bar{y} x_i$$

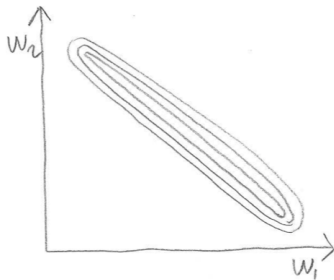


- Which weight, w_1 or w_2 , will receive a larger gradient descent update?
- Which one do you want to receive a larger update?
- Note: the figure vastly *understates* the narrowness of the ravine!

Ill-conditioned curvature: normalization

- Or consider the following dataset:

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
\vdots	\vdots	\vdots



III-conditioned curvature: normalization

- To avoid these problems, it's a good idea to center your inputs to zero mean and unit variance, especially when they're in arbitrary units (feet, seconds, etc.).

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

- Hidden units may have non-centered activations, and this is harder to deal with.
 - One trick: replace logistic units (which range from 0 to 1) with tanh units (which range from -1 to 1)
 - A recent method called **batch normalization** explicitly centers each hidden activation. It often speeds up training by 1.5-2x, and it's available in all the major neural net frameworks.

Momentum

- Unfortunately, even with these normalization tricks, ill-conditioned curvature is a fact of life. We need algorithms that are able to deal with it.
- **Momentum** is a simple and highly effective method. Imagine a hockey puck on a frictionless surface (representing the cost function). It will accumulate momentum in the downhill direction:

$$\begin{aligned} \mathbf{p} &\leftarrow \mu \mathbf{p} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \mathbf{p} \end{aligned}$$

- α is the learning rate, just like in gradient descent.
- μ is a damping parameter. It should be slightly less than 1 (e.g. 0.9 or 0.99). Why not exactly 1?
 - If $\mu = 1$, conservation of energy implies it will never settle down.

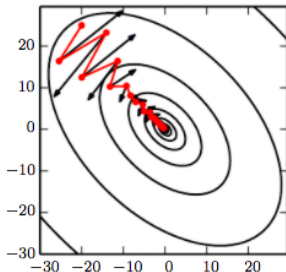
Momentum

- In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.
- In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.
- If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

$$-\frac{\alpha}{1 - \mu} \cdot \frac{\partial \mathcal{J}}{\partial \theta}$$

This suggests if you increase μ , you should lower α to compensate.

- Momentum sometimes helps a lot, and almost never hurts.



Ravines

- Even with momentum and normalization tricks, narrow ravines are still one of the biggest obstacles in optimizing neural networks.
- Empirically, the curvature can be many orders of magnitude larger in some directions than others!
- An area of research known as **second-order optimization** develops algorithms which explicitly use curvature information (second derivatives), but these are complicated and difficult to scale to large neural nets and large datasets.
- There is an optimization procedure called **Adam** which uses just a little bit of curvature information and often works much better than gradient descent. It's available in all the major neural net frameworks.

RMSprop and Adam

- Recall: Gradient descent takes large steps in directions of high curvature and small steps in directions of low curvature.
- **RMSprop** is a variant of GD which rescales each coordinate of the gradient to have norm 1 on average. It does this by keeping an exponential moving average s_j of the squared gradients.
- The following update is applied to each coordinate j independently:

$$s_j \leftarrow (1 - \gamma)s_j + \gamma \left[\frac{\partial \mathcal{J}}{\partial \theta_j} \right]^2$$
$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{s_j} + \epsilon} \frac{\partial \mathcal{J}}{\partial \theta_j}$$

- If the eigenvectors of the Hessian are axis-aligned (dubious assumption), then RMSprop can correct for the curvature. In practice, it typically works slightly better than SGD.
- **Adam** = RMSprop + momentum
- Both optimizers are included in TensorFlow, Pytorch, etc.

Recap

- We've seen how to analyze the typical phenomena in optimization:
 - **Local minima:** neural nets are not convex.
 - **Saddle points:** Hessian has both positive and negative eigenvalues. Occurs when there are weight symmetries upon initialization.
 - **Plateaux:** Jacobian close to zero, e.g. dead neurons.
 - **Ill-conditioned curvature (ravines):** Hessian has extremely large and very small positive eigenvalues. Affect the largest possible learning rate before divergence.
- You will likely encounter some of these problems when training neural nets.
- This lecture helps understanding the causes of these phenomena. We will discuss the workarounds in a future lecture.