Training DNNs: Basic Methods and Tricks

Ju Sun
Computer Science & Engineering
University of Minnesota, Twin Cities

February 19, 2026

Supervised learning as data-fitting

- **Step 1** Gather training set $(\boldsymbol{x}_1, \boldsymbol{y}_1)$, . . . , $(\boldsymbol{x}_n, \boldsymbol{y}_m)$
- **Step 2** Choose a NN with k neurons, so that there exist weights $(\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)$ to ensure $\boldsymbol{y}_i \approx \{\text{NN}\,(\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)\}\,(\boldsymbol{x}_i)$, $\forall i$
- **Step 3** Set up a loss function ℓ
- **Step 4** Find weights $(\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)$ to minimize the average loss

$$\frac{1}{m}\sum_{i=1}^{m} \ell\left[\boldsymbol{y}_i, \{\text{NN}\,(\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)\}\,(\boldsymbol{x}_i)\right]$$

Three fundamental questions in DL

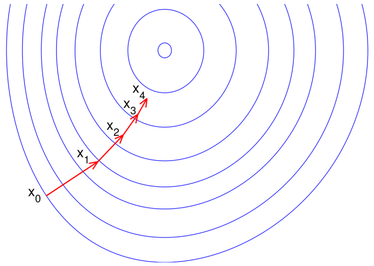
- **Approximation**: is it powerful, i.e., the $\mathcal{H}$ large enough for all possible weights?
- **Optimization**: how to solve

$$\min_{\boldsymbol{w}_i' s} \frac{1}{m}\sum_{i=1}^{m} \ell\left[\boldsymbol{y}_i, \{\text{NN}\,(\boldsymbol{w}_1, \dots, \boldsymbol{w}_k)\}\,(\boldsymbol{x}_i)\right]$$

- **Generalization**: does the learned NN work well on "similar" data? (CSCI5525, and Deep Learning Theory)

Basics of numerical optimization

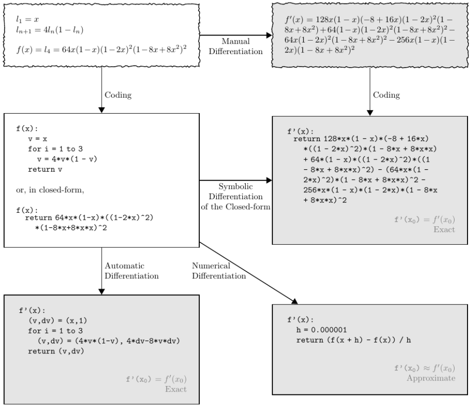
- 1st and 2nd optimality conditions
- iterative methods



Credit: aria42.com

- gradient descent
- Newton's method
- momentum methods
- quasi-Newton methods
- coordinate descent
- conjugate gradient methods
- trust-region methods
- etc

Computing derivatives



Manual Differentiation
Coding
Coding
Symbolic Differentiation of the Closed-form
Automatic Differentiation
Numerical Differentiation

$l_1 = x$
$l_{n+1} = 4l_n(1-l_n)$

$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$

$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$

```
f(x):
   v = x
   for i = 1 to 3
     v = 4*v*(1 - v)
   return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

```
f'(x):
 return 128*x*(1 - x)*(-8 + 16*x)
   *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
   + 64*(1 - x)*((1 - 2*x)^2)*((1
   - 8*x + 8*x*x)^2) - (64*x*(1 -
   2*x)^2)*(1 - 8*x + 8*x*x)^2 -
   256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
   + 8*x*x)^2
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
   (v,dv) = (x,1)
   for i = 1 to 3
     (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
   return (v,dv)
```

f'(x_0) $= f'(x_0)$
Exact

```
f'(x):
   h = 0.000001
   return (f(x + h) - f(x)) / h
```

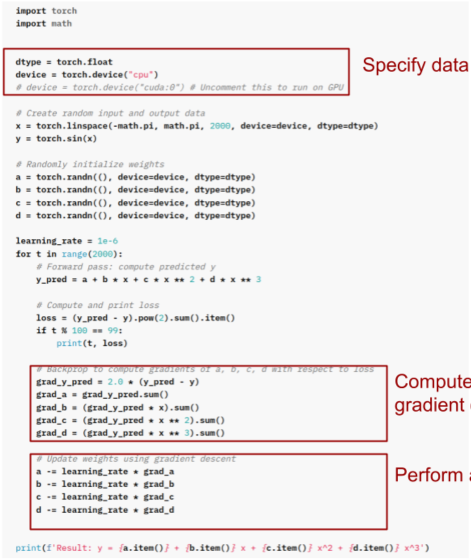
f'(x_0) $\approx f'(x_0)$
Approximate

Credit: [Baydin et al., 2017]

- Analytic differentiation (by hand or by software)
- Finite difference approximation
- Automatic/Algorithmic differentiation (AD)

A simple example

train model $y = a + bx + cx^2 + dx^3$ using plain pytorch



```
import torch
import math

dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# Create random input and output data
x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
y = torch.sin(x)

# Randomly initialize weights
a = torch.randn((), device=device, dtype=dtype)
b = torch.randn((), device=device, dtype=dtype)
c = torch.randn((), device=device, dtype=dtype)
d = torch.randn((), device=device, dtype=dtype)

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum().item()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights using gradient descent
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

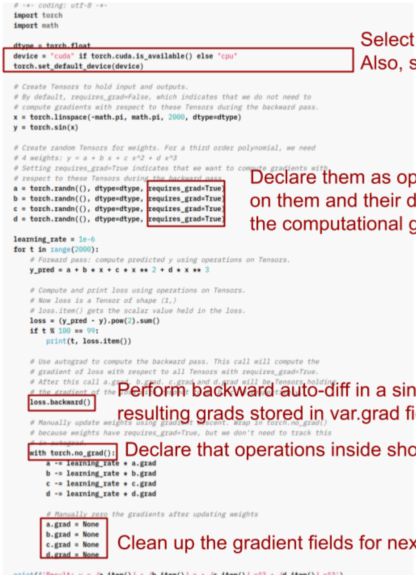
Specify data type and device

Compute numerical gradient by analytical gradient or auto-differentiation

Perform a gradient descent step

A simple example

train model $y = a + bx + cx^2 + dx^3$ using PyTorch with autodiff (backward)



```python
# -*- coding: utf-8 -*-
import torch
import math

dtype = torch.float
device = "cuda" if torch.cuda.is_available() else "cpu"
torch.set_default_device(device)

# Create Tensors to hold input and outputs.
# By default, requires_grad=False, which indicates that we do not need to
# compute gradients with respect to these Tensors during the backward pass.
x = torch.linspace(-math.pi, math.pi, 2000, dtype=dtype)
y = torch.sin(x)

# Create random Tensors for weights. For a third order polynomial, we need
# 4 weights: y = a + b x + c x^2 + d x^3
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
a = torch.randn((), dtype=dtype, requires_grad=True)
b = torch.randn((), dtype=dtype, requires_grad=True)
c = torch.randn((), dtype=dtype, requires_grad=True)
d = torch.randn((), dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Select device, prioritizing GPU if available.
Also, set default device

Declare them as optimization variables. All operations on them and their descents will be tracked and built into the computational graph, unless told otherwise

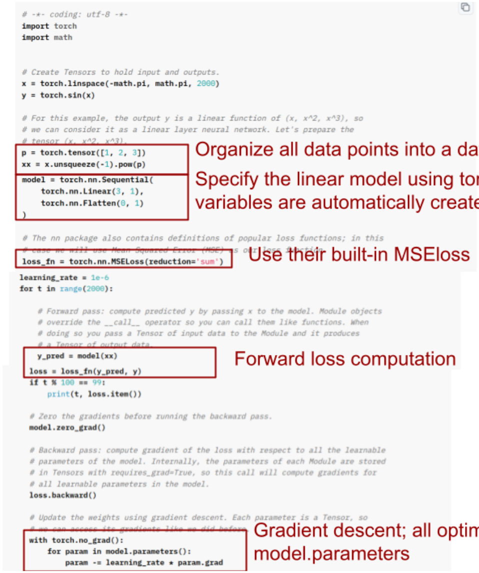
Perform backward auto-diff in a single line, resulting grads stored in var.grad fields

Declare that operations inside shouldn't be tracked

Clean up the gradient fields for next iteration

A simple example

train model $y = [a\ b\ c\ d]^\intercal [1\ x\ x^2\ x^3]$ using PyTorch with `torch.nn`



```
# -*- coding: utf-8 -*-
import torch
import math


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# For this example, the output y is a linear function of (x, x^2, x^3), so
# we can consider it as a linear layer neural network. Let's prepare the
# tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)
```

Organize all data points into a data tensor

```
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
```

Specify the linear model using torch.nn module; optimization variables are automatically created and initialized

```
# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')
```

Use their built-in MSEloss

```
learning_rate = 1e-6
for t in range(2000):

    # Forward pass: compute predicted y by passing x to the model. Module objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
```

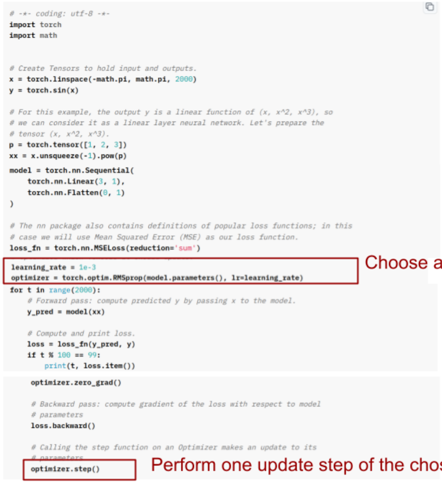
Forward loss computation

```
    if t % 100 == 99:
        print(t, loss.item())

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the learnable
    # parameters of the model. Internally, the parameters of each Module are stored
    # in Tensors with requires_grad=True, so this call will compute gradients for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

Gradient descent; all optimization variables stored in model.parameters

A simple example

train model $y = [a\ b\ c\ d]^\intercal [1\ x\ x^2\ x^3]$ using PyTorch with `torch.nn` and a built-in optimizer



```python
# -*- coding: utf-8 -*-
import torch
import math


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# For this example, the output y is a linear function of (x, x^2, x^3), so
# we can consider it as a linear layer neural network. Let's prepare the
# tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

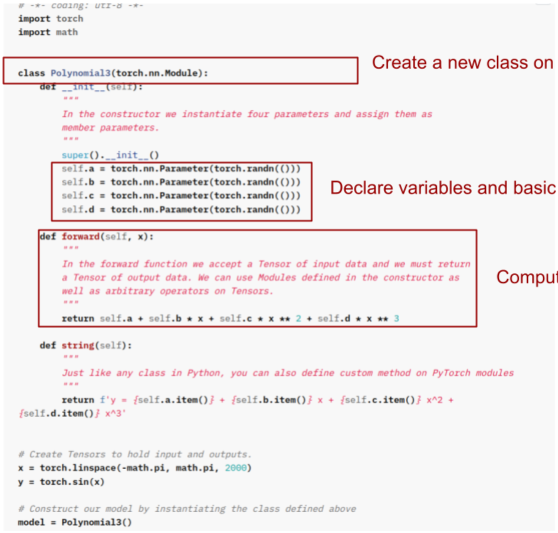
Choose an internal optimizer/solver

Perform one update step of the chosen optimizer

PyTorch optimizers: https://pytorch.org/docs/stable/optim.html

A simple example

train model $y = [a\ b\ c\ d]^\intercal [1\ x\ x^2\ x^3]$ using PyTorch with customized model class based on `torch.nn`



```
# -*- coding: utf-8 -*-
import torch
import math


class Polynomial3(torch.nn.Module):
    def __init__(self):
        """
        In the constructor we instantiate four parameters and assign them as
        member parameters.
        """
        super().__init__()
        self.a = torch.nn.Parameter(torch.randn(()))
        self.b = torch.nn.Parameter(torch.randn(()))
        self.c = torch.nn.Parameter(torch.randn(()))
        self.d = torch.nn.Parameter(torch.randn(()))

    def forward(self, x):
        """
        In the forward function we accept a Tensor of input data and we must return
        a Tensor of output data. We can use Modules defined in the constructor as
        well as arbitrary operators on Tensors.
        """
        return self.a + self.b * x + self.c * x ** 2 + self.d * x ** 3

    def string(self):
        """
        Just like any class in Python, you can also define custom method on PyTorch modules
        """
        return f'y = {self.a.item()} + {self.b.item()} x + {self.c.item()} x^2 +
{self.d.item()} x^3'


# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Construct our model by instantiating the class defined above
model = Polynomial3()
```

Create a new class on top of torch.nn class

Declare variables and basic building blocks

Compute forward loss

Build an MLP

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Please work through `Introduction to PyTorch`
https://pytorch.org/tutorials/beginner/basics/intro.html
by yourself

Ready to optimize DNNs!?

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

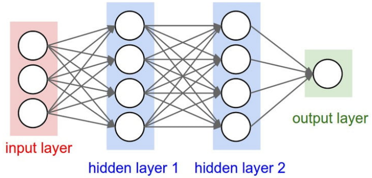
Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

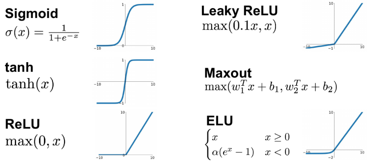
Suggested reading

Set up the problem

DNN



activation function



Credit: Stanford CS231N

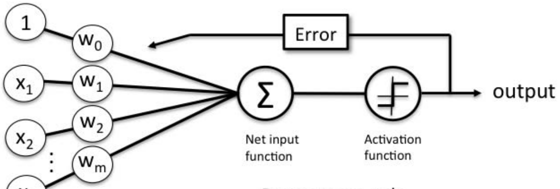
$$\min_{\boldsymbol{W}} \sum_i \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

– Which activation at the hidden nodes?

– Which activation at the output node?

– Which ℓ?

Which activation at the hidden nodes?



Perceptron rule.

Is the $\operatorname{sign}(\cdot)$ activation good for derivative-based optimization?

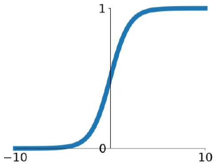
$$\nabla_{\boldsymbol{w}} \ell\left(\operatorname{sign}\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right), y\right) = \ell'\left(\operatorname{sign}\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right), y\right) \operatorname{sign}'\left(\boldsymbol{w}^\intercal \boldsymbol{x}\right) \boldsymbol{x} = \mathbf{0}$$

almost everywhere (But why the classic Perceptron algorithm converges?)

Desiderata for activation:

- Differentiable or almost everywhere differentiable
- Nonzero derivatives (almost) everywhere
- Cheap to compute

Sigmoid and hypertangent

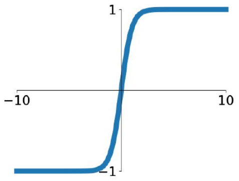


Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

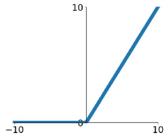
- Differentiable? **Yes!**
- Nonzero derivatives? **Yes and No!** What happens for large positive and negative inputs?
- Cheap? $\exp(\cdot)$ is relatively expensive

What about $\tanh$?



tanh(x)

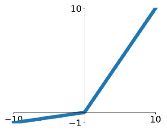
ReLU and friends



ReLU
(Rectified Linear Unit)

$$\sigma(x) = \max(0, x)$$

- Differentiable? **Yes!** (almost everywhere)
- Nonzero derivatives? **Yes and No!** What happens for $x < 0$?
- Cheap? **Yes!**

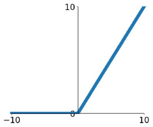


Leaky ReLU

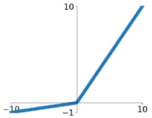
$$\sigma(x) = \max(\alpha x, x) \quad \text{(e.g., } \alpha = 0.01\text{)}$$

- Differentiable? **Yes!** (almost everywhere)
- Nonzero derivatives? **Yes!** (almost everywhere)
- Cheap? **Yes!**

ReLU and friends

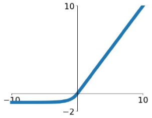


ReLU
(Rectified Linear Unit)



Leaky ReLU

Exponential Linear Units (ELU)



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- ReLU and Leaky ReLU are the most popular
- $\tanh$ less preferred but okay; sigmoid should be avoided

https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Which activation at output node?

depending on the desired output

- unbounded scalar/vector output (e.g. , regression): identity activation
- binary classification with 0 or 1 output: e.g., sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ or $\sigma(x) = \frac{1}{2}(\sin(x) + 1)$
- multiclass classification: labels into vectors via one-hot encoding

$$L_k \Longrightarrow [\underbrace{0, \dots, 0,}_{k-1\, 0's}, 1, \underbrace{0, \dots, 0}_{n-k\, 0's}]^\intercal$$

  e.g., softmax activation: $\boldsymbol{z} \mapsto \left[\frac{e^{z_1}}{\sum_j e^{z_j}}, \dots, \frac{e^{z_p}}{\sum_j e^{z_j}}\right]^\intercal$.
- discrete probability distribution: softmax
- for more involved constraints: embedded projection/relaxation layers [Min and Azizan, 2024, Chu et al., 2026]

Which loss?

Which ℓ to choose? Make it differentiable, or almost so

- **regression**: $\|\cdot\|_2^2$ (common, `torch.nn.MSELoss`), $\|\cdot\|_1$ (for robustness, `torch.nn.L1Loss`), etc
- **binary classification**: encoder the classes as $\{0, 1\}$, $\|\cdot\|_2^2$ or cross-entropy: $\ell(y, \hat{y}) = y \log \hat{y} - (1 - y) \log(1 - \hat{y})$ (min at $\hat{y} = y$, `torch.nn.BCELoss`)
- **multiclass classification** based on one-hot encoding and softmax activation: $\|\cdot\|_2^2$ or cross-entropy: $\ell(\boldsymbol{y}, \widehat{\boldsymbol{y}}) = -\sum_i y_i \log \widehat{y_i}$ (min at $\boldsymbol{y} = \widehat{\boldsymbol{y}}$, `torch.nn.CrossEntropyLoss`)
    * **label smoothing**: one-hot encoding makes all y_i's zero except for the target class, but $y_i = 0 \implies \nabla_{\boldsymbol{w}} y_i \log \widehat{y_i} = 0 \implies$ no update contributed from y_i.
      Remedy: relax ... change $[0, \ldots, 0, 1, 0, \ldots, 0]^\intercal$ into $[\varepsilon, \ldots, \varepsilon, 1 - (m-1)\varepsilon, \varepsilon, \ldots, \varepsilon]^\intercal$ for a small ε
- **difference between distributions**: Kullback-Leibler divergence loss (`torch.nn.KLDivLoss`) or Wasserstein metric

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Recall: framework of line-search methods

A generic line search algorithm

```
Input: initialization $\boldsymbol{x}_0$, stopping criterion (SC), $k = 1$
1: while SC not satisfied do
2:     choose a direction $\boldsymbol{d}_k$
3:     decide a step size $t_k$
4:     make a step: $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + t_k \boldsymbol{d}_k$
5:     update counter: $k = k + 1$
6: end while
```

Four questions:

- How to choose direction $\boldsymbol{d}_k$?
- How to choose step size t_k?
- Where to initialize?
- When to stop?

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

From deterministic to stochastic optimization

Recall our optimization problem:

$$\min_{\boldsymbol{W}} \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

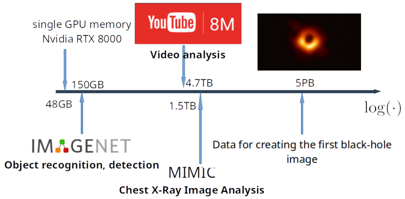
What happens when m is large, i.e., in the "big data" regime?

Blessing: assume $(\boldsymbol{x}_i, \boldsymbol{y}_i)$'s are iid, then

$$\frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}} \ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$$

by the law of large numbers. Large $m \approx$ good generalization!

Curse: storage and computation



- **storage**: $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}$ typically loaded onto GPU/TPU for parallel computing—loading whole dataset not feasible
- **computation**: each iteration costs at least $O(mn)$, where n is #(opt variables)—both can be large for training DNNs!

From deterministic to stochastic optimization

How to get around for large m?

stochastic optimization (stochastic = random)

Idea: use a small batch of data samples to approximate quantities of interest

- gradient: $\frac{1}{m}\sum_{i=1}^{m}\nabla_{\boldsymbol{W}}\ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}}\nabla_{\boldsymbol{W}}\ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$

  approximated by **stochastic gradient**:

  $$\frac{1}{|J|}\sum_{j\in J}\nabla_{\boldsymbol{W}}\ell\left(\boldsymbol{y}_j, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_j\right)\right)$$

  for a random subset $J \subset \{1, \dots, m\}$, where $|J| \ll m$
- Hessian: $\frac{1}{m}\sum_{i=1}^{m}\nabla^2_{\boldsymbol{W}}\ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) \to \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}}\nabla^2_{\boldsymbol{W}}\ell\left(\boldsymbol{y}, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}\right)\right)$

  approximated by **stochastic Hessian**:

  $$\frac{1}{|J|}\sum_{j\in J}\nabla^2_{\boldsymbol{W}}\ell\left(\boldsymbol{y}_j, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_j\right)\right)$$

  for a random subset $J \subset \{1, \dots, m\}$, where $|J| \ll m$

... justified by the law of large numbers

Stochastic gradient descent (SGD)

In general (i.e., not only for DNNs), suppose we want to solve

$$\min_{\boldsymbol{w}} \; F(\boldsymbol{w}) \doteq \frac{1}{m} \sum\nolimits_{i=1}^{m} f(\boldsymbol{w}; \boldsymbol{\xi}_i) \qquad \boldsymbol{\xi}_i\text{'s are data samples}$$

idea: replace gradient with a stochastic gradient in each step of GD

Stochastic gradient descent (SGD)

```
Input: initialization $\boldsymbol{w}_0$, stopping criterion (SC), $k = 1$
1: while SC not satisfied do
2:     sample a random subset $J_k \subset \{0, \dots, m-1\}$
3:     calculate the stochastic gradient $\widehat{\boldsymbol{g}_k} \doteq \frac{1}{|J_k|} \sum_{j \in J_k} \nabla_{\boldsymbol{w}} f(\boldsymbol{w}_{k-1}; \boldsymbol{\xi}_j)$
4:     decide a step size $t_k$
5:     make a step: $\boldsymbol{w}_k = \boldsymbol{w}_{k-1} - t_k \widehat{\boldsymbol{g}_k}$
6:     update counter: $k = k + 1$
7: end while
```

- J_k is redrawn in each iteration
- Traditional SGD: $|J_k| = 1$. The version presented is also called **mini-batch gradient descent**

What's an epoch?

- Canonical SGD: sample a random subset $J_k \subset \{1, \ldots, m\}$ each iteration—sampling with replacement
- Practical SGD: shuffle the training set, and take a consecutive batch of size B (called **batch size**) each iteration—sampling without replacement

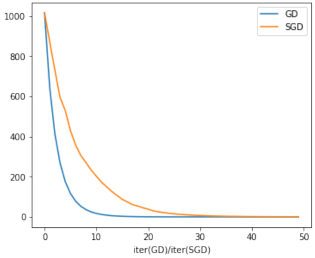
one pass of the shuffled training set is called one **epoch**.

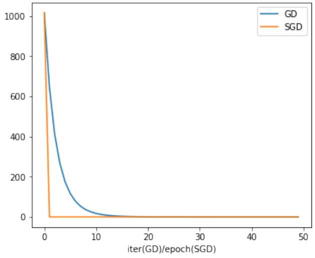
Practical stochastic gradient descent (SGD)

```
Input: init. $\boldsymbol{w}_0$, SC, batch size $B$, iteration counter $k = 1$, epoch counter $\ell = 1$
 1: while SC not satisfied do
 2:     permute the index set $\{0, \cdots, m\}$ and divide it into batches of size $B$
 3:     for $i \in \{1, \ldots, \#\text{batches}\}$ do
 4:         calculate the stochastic gradient $\widehat{\boldsymbol{g}_k}$ based on the $i^{th}$ batch
 5:         decide a step size $t_k$
 6:         make a step: $\boldsymbol{w}_k = \boldsymbol{w}_{k-1} - t_k \widehat{\boldsymbol{g}_k}$
 7:         update iteration counter: $k = k + 1$
 8:     end for
 9:     update epoch counter: $\ell = \ell + 1$
10: end while
```

GD vs. SGD

Consider $\min_{\boldsymbol{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$, where $\boldsymbol{X} \in \mathbb{R}^{10000 \times 500}$, $\boldsymbol{y} \in \mathbb{R}^{10000}$, $\boldsymbol{w} \in \mathbb{R}^{500}$





- By iteration: GD is faster
- By iter(GD)/epoch(SGD): SGD is faster
- Remember, cost of one epoch of SGD $\approx$ cost of one iteration of GD!

SGD is quicker to find a medium-accuracy solution with lower cost, which suffices for most purposes in machine learning [Bottou and Bousquet, 2008].

Step size (learning rate) for SGD

Recall the recommended step size rule for GD: **back-tracking line search**

key idea: $F\left(\boldsymbol{w} - t\nabla F\left(\boldsymbol{w}\right)\right) - F\left(\boldsymbol{w}\right) \approx -ct\left\|\nabla F\left(\boldsymbol{w}\right)\right\|^2$ for a certain $c \in (0, 1)$

Shall we do it for SGD? No, but why?

- SGD tries to avoid the m factor in computing the full gradient $\nabla_{\boldsymbol{w}} F\left(\boldsymbol{w}\right) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{w}} f\left(\boldsymbol{w}; \boldsymbol{\xi}_i\right)$, i.e., reducing m to B (batch size)
- But computing $F\left(\boldsymbol{w}\right) = \frac{1}{m}\sum_{i=1}^{m} f\left(\boldsymbol{w}; \boldsymbol{\xi}_i\right)$ or $F\left(\boldsymbol{w} - t\widehat{\boldsymbol{g}}\right) = \frac{1}{m}\sum_{i=1}^{m} f\left(\boldsymbol{w} - t\widehat{\boldsymbol{g}}; \boldsymbol{\xi}_i\right)$ brings back the m factor; similarly for ∇F
- What about computing approximations to the objective values based on small batches also? Approximation errors for F and ∇F may ruin the stability of the Taylor criterion

Step size (learning rate, or LR) for SGD

Classical theory for SGD on convex problems requires

$$\sum_k t_k = \infty, \quad \sum_k t_k^2 < \infty.$$

Practical implementation: diminishing step size/LR, e.g.,

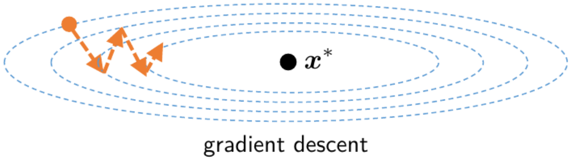
- $1/k$ **delay**: $t_k = \alpha/(1+\beta k)$, α, β: tunable parameters, k: iteration index
- **exponential delay**: $t_k = \alpha e^{-\beta k}$, α, β: tunable parameters, k: iteration index
- **staircase delay**: start from t_0, divide it by a factor (e.g., 5 or 10) every L (say, 10) epochs—popular in practice. Some heuristic variants: (`lr_scheduler.ReduceLROnPlateau`)
  - watch the validation error and decrease the LR when it stagnates
  - watch the objective and decrease the LR when it stagnates

check out `torch.optim.lr_scheduler` in PyTorch! https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

Beyond the vanilla SGD

- Momentum/acceleration methods
- SGD with adaptive learning rates
- Stochastic 2nd order methods

Why momentum?



Credit: Princeton ELE522

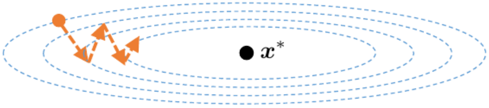
- GD is cheap ($O(n)$ per step) but overall convergence sensitive to conditioning
- Newton's convergence is not sensitive to conditioning but expensive ($O(n^3)$ per step)

A cheap way to achieve faster convergence? Answer: using historic information

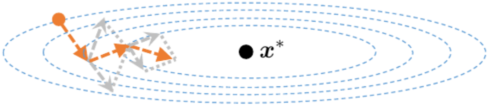
Heavy ball method

In physics, a heavy object has a large inertia/momentum—resistance to change in velocity.

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k) + \beta_k \underbrace{(\boldsymbol{x}_k - \boldsymbol{x}_{k-1})}_{\textbf{momentum}} \quad \text{due to Polyak}$$



gradient descent



heavy-ball method

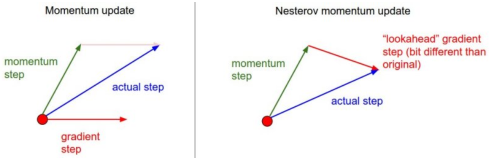
Credit: Princeton ELE522

History helps to smooth out the zig-zag path!

Nesterov's accelerated gradient methods

due to Y. Nesterov

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \beta_k \left(\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\right) - \alpha_k \nabla f\left(\boldsymbol{x}_k + \beta_k \left(\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\right)\right)$$



Credit: Stanford CS231N

$$\text{HB} \begin{cases} x_{\text{ahead}} = x + \beta(x - x_{\text{old}}), \\ x_{\text{new}} = x_{\text{ahead}} - \alpha \nabla f(x). \end{cases} \quad \text{Nesterov} \begin{cases} x_{\text{ahead}} = x + \beta(x - x_{\text{old}}), \\ x_{\text{new}} = x_{\text{ahead}} - \alpha \nabla f(x_{\text{ahead}}). \end{cases}$$

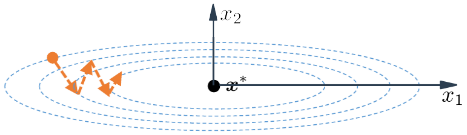
SGD with momentum/acceleration: replace the gradient term ∇f by the stochastic gradient $\widehat{\boldsymbol{g}}$ based on small batches

check out `torch.optim.SGD` at (their convention slightly differs from here)

https://pytorch.org/docs/stable/optim.html#torch.optim.SGD

Why SGD with adaptive learning rate?

Recall the struggle of GD on elongated functions, e.g., $f(x_1, x_2) = x_1^2 + 4x_2^2$



gradient descent

- (Quasi-)Newton's method: take the full curvature info, but expensive
- Momentum methods: use historic direction(s) to cancel out wiggles

Another heuristic remedy: balance out movements in all coordinate directions.
Suppose $\boldsymbol{g}$ is the (stochastic) gradient, for all i,

divide g_i by historic gradient magnitudes in the i^{th} coordinate

Benefit: coordinate directions always with small (large) derivatives get sped up (slowed down). Think of the above $f(x_1, x_2)$ example!

Method 1: Adagrad

divide g_i by historic gradient magnitudes in the i^{th} coordinate

At the $(k+1)^{th}$ iteration, for all i,

$$x_{i,k+1} = x_{i,k} - t_k \frac{g_{i,k}}{\sqrt{\sum_{j=1}^{k} g_{i,j}^2 + \varepsilon}}$$

or in elementwise notation

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\sum_{j=1}^{k} \boldsymbol{g}_j^2 + \boldsymbol{\varepsilon}}}$$

Write $\boldsymbol{s}_k \doteq \sum_{j=1}^{k} \boldsymbol{g}_j^2$. Note that $\boldsymbol{s}_k = \boldsymbol{s}_{k-1} + \boldsymbol{g}_k^2$. So only need to incrementally update the $\boldsymbol{s}_k$ sequence, which is cheap

In PyTorch, `torch.optim.Adagrad`

https://pytorch.org/docs/stable/optim.html#torch.optim.Adagrad

Method 2: RMSprop

Adagrad:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \boldsymbol{\varepsilon}}} \quad \text{with } \boldsymbol{s}_k \doteq \sum_{j=1}^{k} \boldsymbol{g}_j^2 .$$

$$\text{update equation for } \boldsymbol{s}_k : \boldsymbol{s}_k = \boldsymbol{s}_{k-1} + \boldsymbol{g}_k^2$$

Problems:

- Magnitudes in $\boldsymbol{s}_k$ becomes larger when k grows, and hence movements $t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}}$ become small when k is large.
- Remote history may not be relevant

Solution: **RMSprop**—gradually phase out the history. For some $\beta \in (0, 1)$

$$\boldsymbol{s}_k = \beta \boldsymbol{s}_{k-1} + (1-\beta)\, \boldsymbol{g}_k^2 \Longleftrightarrow \boldsymbol{s}_k = (1-\beta) \left(\boldsymbol{g}_k^2 + \beta \boldsymbol{g}_{k-1}^2 + \beta^2 \boldsymbol{g}_{k-2}^2 + \dots \right)$$

Typical values for β: $0.9, 0.99$. In PyTorch, `torch.optim.RMSprop`

https://pytorch.org/docs/stable/optim.html#torch.optim.RMSprop

Method 3: Adam

Combine RMSprop with momentum methods

$$\begin{aligned} \boldsymbol{m}_k &= \beta_1 \boldsymbol{m}_{k-1} + (1-\beta_1)\, \boldsymbol{g}_k && \text{(combine momentum and stochastic gradient)} \\ \boldsymbol{s}_k &= \beta_2 \boldsymbol{s}_{k-1} + (1-\beta_2)\, \boldsymbol{g}_k^2 && \text{(scaling factor update as in RMSprop)} \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k - t_k \frac{\boldsymbol{m}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}} \end{aligned}$$

- Typical parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon =$ 1e-8.
- In PyTorch, `torch.optim.Adam`
  https://pytorch.org/docs/stable/optim.html#torch.optim.Adam
- Several recent variants: `torch.optim.AdamW`, `torch.optim.SparseAdam`, `torch.optim.Adamax`

Thoughts on adaptive LR methods

- adapting the LR or adapting the (stochastic) gradient? Two views of the same thing ($\odot$ denotes elementwise product)

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \frac{t_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}} \odot \boldsymbol{g}_k \quad \text{vs.} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \frac{\boldsymbol{g}_k}{\sqrt{\boldsymbol{s}_k + \varepsilon}}$$

- adapting the gradient, familiar? What happens in Newton's method?

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \operatorname{diag}\left(\frac{1}{\sqrt{\boldsymbol{s}_k + \varepsilon}}\right) \boldsymbol{g}_k \quad \text{vs.} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - t_k \boldsymbol{H}_k^{-1} \boldsymbol{g}_k.$$

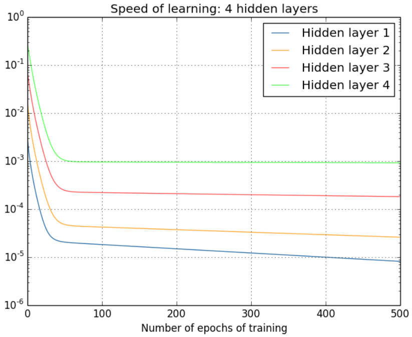
  ... approximate the Hessian (inverse) with a diagonal matrix. So adaptive methods are approximate 2nd order methods, and more faithful approximation possible.

- Learning rate t_k: similar to that for the vanilla SGD, but less sensitive and can be large

Why adaptive methods relevant for DL?

$$F(\boldsymbol{W}_1, \dots, \boldsymbol{W}_k) = \frac{1}{m}\sum_{i=1}^m \ell(\boldsymbol{y}_i, \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \dots (\boldsymbol{W}_1 \boldsymbol{x}_i))))$$

Derivatives for early layers tend to be order of magnitude smaller than those for late layers, i.e., the **gradient vanishing/exploding phenomenon**



See more discussion and explanation in

http://neuralnetworksanddeeplearning.com/chap5.html

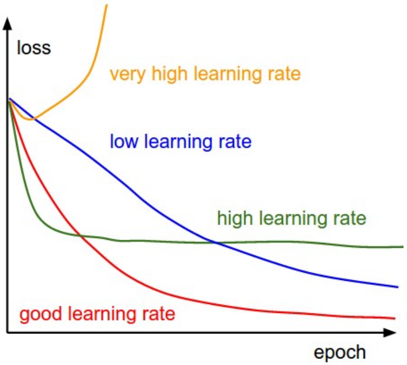
Why adaptive methods relevant for DL?

$$F\left(\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k\right) = \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \ldots (\boldsymbol{W}_1 \boldsymbol{x}_i))\right)\right)$$

- Hypothesis: F has many saddle points and escaping saddle points causes the difficulty of training [Choromanska et al., 2015, Pascanu et al., 2014, Dauphin et al., 2014, Baskerville et al., 2020]
- Adaptive methods can escape saddle points efficiently; see, e.g., [Staib et al., 2020]

visual comparison `https://imgur.com/a/Hqolp`

Diagnosis of LR



Credit: Stanford CS231N

- Low LR always leads to convergence, but takes forever
- Premature flattening is a sign of large LR; premature sloping is a sign of early stopping—increase the number of epochs!
- Remember the starecase LR schedule!

Stochastic 2nd order methods

Recall scalable 2nd order methods

- Quasi-Newton methods, esp. L-BFGS
- Trust-region methods

When #samples is large, use mini batches to estimate any quantities

- stochastic quasi-Newton methods: e.g., [Martens and Grosse, 2015] [Byrd et al., 2016] [Anil et al., 2020] [Roosta-Khorasani and Mahoney, 2018]
- stochastic trust-region methods: e.g., [Curtis and Shi, 2019], [Chauhan et al., 2018]

`https://github.com/hjmshi/PyTorch-LBFGS`
`https://pytorch-optimizers.readthedocs.io/en/latest/` (collectes many optimizers not officially supported by PyTorch)

still active area of research. Hardware seems to be the main limiting factor, which is diminishing as GPU memory/compute gets better over years

Outline

Three design choices

Training algorithms

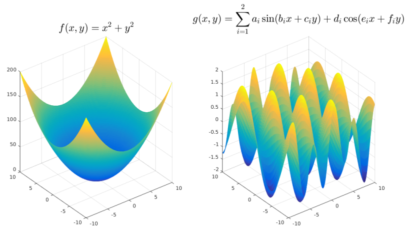
- Which method
- Where to start
- When to stop

Tricks

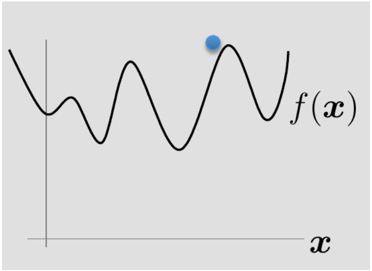
- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Where to initialize? the general picture



convex vs. nonconvex functions



- **Convex**: most iterative methods converge to the global min no matter the initialization
- **Nonconvex**: initialization matters a lot. Common heuristics: random initialization, multiple independent runs
- **Nonconvex**: clever initialization is possible with certain assumptions on the data:

  https://sunju.org/research/nonconvex/

  and sometimes random initialization works!

Where to initialize for DNNs?

$$F(\boldsymbol{W}_1, \dots, \boldsymbol{W}_k) = \frac{1}{m} \sum_{i=1}^m \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \dots (\boldsymbol{W}_1 \boldsymbol{x}_i))\right)\right)$$

– Are there bad initializations? Consider a simple case

$$F(\boldsymbol{W}_1, \boldsymbol{W}_2) = \frac{1}{m} \sum_{i=1}^m \left\| \boldsymbol{y}_i - \boldsymbol{W}_2 \sigma(\boldsymbol{W}_1 \boldsymbol{x}_i) \right\|_2^2$$

$$\nabla_{\boldsymbol{W}_1} F(\boldsymbol{W}_1, \boldsymbol{W}_2) = -\frac{2}{m} \sum_{i=1}^m \left[\boldsymbol{W}_2^\intercal \left(\boldsymbol{y}_i - \boldsymbol{W}_2 \sigma(\boldsymbol{W}_1 \boldsymbol{x}_i)\right) \odot \sigma'(\boldsymbol{W}_1 \boldsymbol{x}_i) \right] \boldsymbol{x}_i^\intercal$$

  * What about $\boldsymbol{W} = \boldsymbol{0}$? $\nabla_{\boldsymbol{W}_1} F = \boldsymbol{0}$—no movement on $\boldsymbol{W}_1$
  * What about very large (small) $\boldsymbol{W}$? Large (small) value & gradient—the problem becomes significant when there are more layers

– Are there principled ways of initialization?

  * random initialization with proper scaling
  * orthogonal initialization

Random initialization

Idea: make all entries in $\boldsymbol{W}$ iid random, and also $\boldsymbol{W}_i$'s and $\boldsymbol{W}_i^\intercal$'s "well behaved"

A reasonable goal: if all entries in $\boldsymbol{v} \in \mathbb{R}^d$ are independent and have zero mean, unit variance, the output $\sigma(\boldsymbol{w}^\intercal \boldsymbol{v}) \in \mathbb{R}$ (i.e., output of a single neuron) has a unit variance.

To seek a specific setting for $\boldsymbol{w} \in \mathbb{R}^d$, suppose $\boldsymbol{w}$ is iid with zero mean and σ is identity. Then:

$$\mathrm{Var}(\boldsymbol{w}^\intercal \boldsymbol{v}) = \mathrm{Var}\left(\sum_i w_i v_i\right) = \sum_i \mathrm{Var}(w_i v_i) = \sum_i \mathrm{Var}(w_i)\,\mathrm{Var}(v_i) = d\mathrm{Var}(w_i).$$

To make $\mathrm{Var}(\boldsymbol{w}^\intercal \boldsymbol{v}) = 1$, we will set $\mathrm{Var}(w_i) = 1/d$.

For $\boldsymbol{W}_i$ with d inputs, set $\boldsymbol{W}_i$ iid zero-mean and $1/d$ variance

Random initialization

$$\boxed{\text{For } \boldsymbol{W}_i \text{ with } d_{\text{in}} \text{ inputs, set } \boldsymbol{W}_i \text{ iid zero-mean and } 1/d_{\text{in}} \text{ variance}}$$

A similar consideration of $\boldsymbol{W}_i^\intercal$ (due to its role in the gradient) also suggests that

$$\boxed{\text{For } \boldsymbol{W}_i \text{ with } d_{\text{out}} \text{ outputs, set } \boldsymbol{W}_i \text{ iid zero-mean and } 1/d_{\text{out}}\text{-variance}}$$

Xavier Initialization: set $\boldsymbol{W}_i \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ iid zero-mean and $\frac{2}{d_{\text{in}}+d_{\text{out}}}$-variance. For example:

- $\boldsymbol{W}_i \sim_{iid} \mathcal{N}\left(0, \frac{2}{d_{\text{in}}+d_{\text{out}}}\right)$ `torch.nn.init.xavier_normal_`
- $\boldsymbol{W}_i \sim_{iid} \text{uniform}\left(-\sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}\right)$ `torch.nn.init.xavier_uniform_`

Random initialization

Recall our derivation assumed σ is identity, which may not be accurate.

For ReLU, assume $\boldsymbol{v}$ iid 0-mean, unit variance, $\boldsymbol{w}$ iid 0-mean, and both $\boldsymbol{v}$, $\boldsymbol{w}$ are symmetric and independent (i,e., $-v$ has the same dist as $\boldsymbol{v}$; similarly for $\boldsymbol{w}$)

$$\mathbb{E}\left[\mathrm{ReLU}^2\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)\right] = \frac{1}{2}\mathbb{E}\left[\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right)^2\right] = \frac{1}{2}\mathrm{Var}\left(\boldsymbol{w}^\intercal \boldsymbol{v}\right) = \frac{1}{2} d \mathrm{Var}\left(w_i\right).$$

Kaiming Initialization (for ReLU): set $\boldsymbol{W}_i \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ iid zero-mean and $\frac{2}{d_{\text{in}}}$-variance. For example:

- $\boldsymbol{W}_i \sim_{iid} \mathcal{N}\left(0, \frac{2}{d_{\text{in}}}\right)$ `torch.nn.init.kaiming_normal_`
- $\boldsymbol{W}_i \sim_{iid} \text{uniform}\left(-\sqrt{\frac{6}{d_{\text{in}}}}, \sqrt{\frac{6}{d_{\text{in}}}}\right)$ `torch.nn.init.kaiming_uniform_`

But it only accounts for d_{in} or d_{out}; a proposed modification: set the variance to $\frac{c}{\sqrt{d_{\text{in}} d_{\text{out}}}}$ for some constant c [Defazio and Bottou, 2019]

Orthogonal initialization

Making all $\boldsymbol{W}_i$'s orthonormal is empirically shown to lead to competitive performance with fewer tricks (covered later). See Sec 4.2 of [Sun, 2019]
`torch.nn.init.orthogonal_`

There is a body of research proposing constraining/regularizing $\boldsymbol{W}_i$'s to be orthonormal, e.g., [Arjovsky et al., 2016, Bansal et al., 2018, Lezcano-Casado and Martínez-Rubio, 2019, Li et al., 2020]

See also the modified PyTorch package that allows manifold constraints
https://github.com/mctorch/mctorch
and the NCVX package that can handle general constrained deep learning
https://ncvx.org/

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

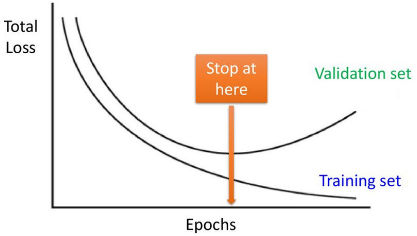
Suggested reading

When to stop in training DNNs?

Recall that a natural stopping criterion for general GD is $\|\nabla f(\boldsymbol{w})\| \le \varepsilon$ for a small ε. Is this good when training DNNs?

- Computing $\nabla f(\boldsymbol{w})$ each iterate is expensive (recall why GD into SGD)
- Stochastic gradient is **noisy**—norm at a true critical point may be large
- Non-differentiable objectives are common in deep learning

A practical/pragmatic stopping strategy for supervised problems: **early stopping**



... periodically check the validation error and stop when it doesn't improve

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Recap

Training DNNs

$$\min_{\boldsymbol{W}} \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

- What methods? Mini-batch stochastic optimization due to large m
  * SGD (with momentum), Adagrad, RMSprop, Adam
  * diminishing LR (1/t, exp delay, staircase delay)
- Where to start?
  * Xavier init., Kaiming init., orthogonal init.
- When to stop?
  * early stopping: stop when validation error doesn't improve

Now: **additional tricks/heuristics that improve**

- **convergence speed**
- **task-specific (e.g., classification, regression, generation) performance**

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Why scaling matters?

Consider a ML objective: $\min_{\boldsymbol{w}} \; f(\boldsymbol{w}) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)$, e.g.,

- Least-squares (LS): $\min_{\boldsymbol{w}} \; \frac{1}{m}\sum_{i=1}^{m} \|y_i - \boldsymbol{w}^\intercal \boldsymbol{x}_i\|_2^2$
- Logistic regression: $\min_{\boldsymbol{w}} \; -\frac{1}{m}\sum_{i=1}^{m} \left[y_i \boldsymbol{w}^\intercal \boldsymbol{x}_i - \log\left(1 + e^{\boldsymbol{w}^\intercal \boldsymbol{x}_i}\right)\right]$
- Shallow NN prediction: $\min_{\boldsymbol{w}} \; \frac{1}{m}\sum_{i=1}^{m} \|y_i - \sigma(\boldsymbol{w}^\intercal \boldsymbol{x}_i)\|_2^2$

Gradient: $\nabla_{\boldsymbol{w}} f = \frac{1}{m}\sum_{i=1}^{m} \ell'(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)\, \boldsymbol{x}_i$.

- What happens when coordinates (i.e., features) of $\boldsymbol{x}_i$ have different orders of magnitude? Partial derivatives have different orders of magnitudes $\Longrightarrow$ slow convergence of vanilla GD (recall why adaptive grad methods)

Hessian: $\nabla^2_{\boldsymbol{w}} f = \frac{1}{m}\sum_{i=1}^{m} \ell''(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)\, \boldsymbol{x}_i \boldsymbol{x}_i^\intercal$.

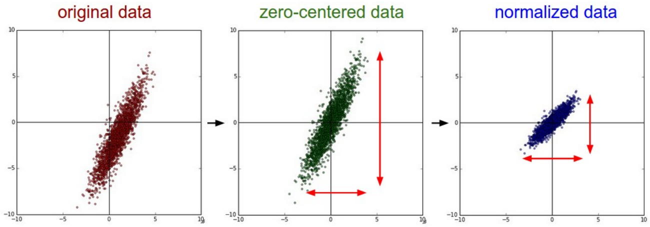
- Suppose the off-diagonal elements of $\boldsymbol{x}_i \boldsymbol{x}_i^\intercal$ are relatively small (e.g., when features are "independent").
- What happens when coordinates (i.e., features) of $\boldsymbol{x}_i$ have different orders of magnitude? Conditioning of $\nabla^2_{\boldsymbol{w}} f$ is bad, i.e., f is elongated

Fix the scaling: first idea

Normalization: make each feature zero-mean and unit variance, i.e., make each feature/column of $\boldsymbol{X}$ zero-mean and unit variance, i.e.

$$\boldsymbol{X}' = \frac{\boldsymbol{X} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \quad (\boldsymbol{\mu}\text{—col means, } \boldsymbol{\sigma}\text{—col std, broadcasting applies})$$

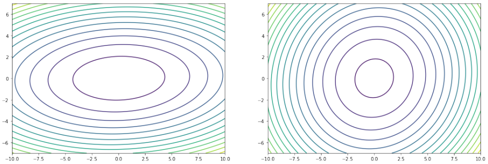
```
X = (X - X.mean(axis=0))/X.std(axis=0)
```



Credit: Stanford CS231N

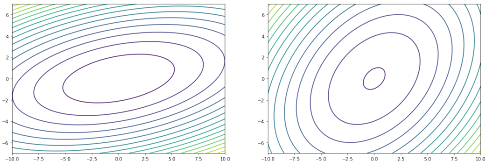
Fix the scaling: first idea

For LS, works well when features are approximately independent



before vs. after the normalization

For LS, works not so well when features are highly dependent.



before vs. after the normalization

How to remove the feature dependency?

Fix the scaling: second idea

PCA and whitening

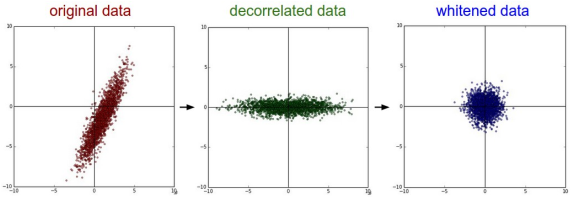
PCA, i.e., zero-center and rotate the data to align principal directions to coordinate directions

```
X -= X.mean(axis=0) #centering
U, S, VT = np.linalg.svd(X, full_matrices=False)
Xrot = X@VT.T #rotate/decorrelate the data
```

(math: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\intercal$, then $\boldsymbol{X}\boldsymbol{V} = \boldsymbol{U}\boldsymbol{S}$)

Whitening: PCA + normalize the coordinates by singular values

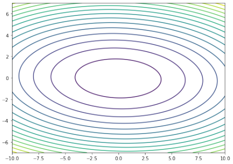
```
Xwhite = Xrot/(S+eps)
```
# (math: $\boldsymbol{X}_{\text{white}} = \boldsymbol{U}$)

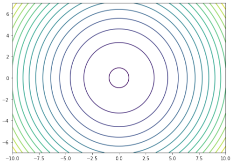


Credit: Stanford CS231N

Fix the scaling: second idea

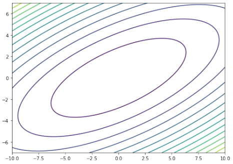
For LS, works well when features are approximately independent

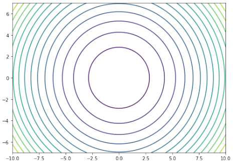




before vs. after the whitening

For LS, also works well when features are highly dependent.





before vs. after the whitening

In DNNs practice

fixing the feature scaling makes the landscape "nicer"—derivatives and curvatures in all directions are roughly even in magnitudes. So for DNNs,

- Preprocess the input data
  - zero-center
  - normalization
  - PCA or whitening (less common)
- But recall our model objective $\min_{\boldsymbol{w}} \; f(\boldsymbol{w}) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell(\boldsymbol{w}^\intercal \boldsymbol{x}_i; y_i)$ vs. DL objective

$$\min_{\boldsymbol{W}} \; \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1} \dots \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

  - DL objective is much more complex
  - But $\sigma\left(\boldsymbol{W}_k \sigma\left(\boldsymbol{W}_{k-1} \dots \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)$ is a composite version of $\boldsymbol{w}^\intercal \boldsymbol{x}_i$:

$$\boldsymbol{W}_1 \boldsymbol{x}_i, \; \boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right), \; \boldsymbol{W}_3 \sigma\left(\boldsymbol{W}_2 \sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right), \; \dots$$

- **Idea: also process the input data to some/all hidden layers**

Batch normalization

Apply normalization to the input data to some/all hidden layers

- $\sigma(\boldsymbol{W}_k\sigma(\boldsymbol{W}_{k-1}\ldots\sigma(\boldsymbol{W}_1\boldsymbol{x}_i)))$ is a composite version of $\boldsymbol{w}^\intercal\boldsymbol{x}_i$:
$$\boldsymbol{W}_1\boldsymbol{x}_i,\ \boldsymbol{W}_2\sigma(\boldsymbol{W}_1\boldsymbol{x}_i),\ \boldsymbol{W}_3\sigma(\boldsymbol{W}_2\sigma(\boldsymbol{W}_1\boldsymbol{x}_i)),\ \ldots$$
- Apply **normalization** to the outputs of the colored parts based on the statistics of a **mini-batch** of $\boldsymbol{x}_i$'s, e.g.,
$$\boldsymbol{W}_2\underbrace{\sigma(\boldsymbol{W}_1\boldsymbol{x}_i)}_{\doteq\boldsymbol{z}_i}\longrightarrow\boldsymbol{W}_2\underbrace{\mathrm{BN}(\sigma(\boldsymbol{W}_1\boldsymbol{x}_i))}_{\mathrm{BN}(\boldsymbol{z}_i)}$$
- Let $\boldsymbol{z}_i$'s be generated from a mini-batch of $\boldsymbol{x}_i$'s and $\boldsymbol{Z}=\begin{bmatrix}\boldsymbol{z}_1^\intercal\\ \vdots\\ \boldsymbol{z}_{|B|}^\intercal\end{bmatrix}$,
$$\mathrm{BN}(\boldsymbol{Z}_j)=\frac{\boldsymbol{Z}_j-\mu_{\boldsymbol{Z}_j}}{\sigma_{\boldsymbol{Z}_j}}\quad\text{for each } j\text{, i.e., for each neuron/feature.}$$
Flexibity restored by optional scaling γ_j's and shifting β_j's:
$$\mathrm{BN}_{\gamma_j,\beta_j}(\boldsymbol{Z}_j)=\gamma_j\frac{\boldsymbol{Z}_j-\mu_{\boldsymbol{Z}_j}}{\sigma_{\boldsymbol{Z}_j}}+\beta_j\quad\text{for each } j.$$

Here, γ_j's and β's are trainable (optimization) variables!

Batch normalization: implementation details

$$\boldsymbol{W}_2 \underbrace{\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)}_{\doteq \boldsymbol{z}_i} \longrightarrow \boldsymbol{W}_2 \underbrace{\mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)}_{\mathrm{BN}(\boldsymbol{z}_i)} \qquad \mathrm{BN}_{\gamma_j,\beta_j}\left(\boldsymbol{Z}_j\right) = \gamma_j \frac{\boldsymbol{Z}_j - \mu_{\boldsymbol{Z}_j}}{\sigma_{\boldsymbol{Z}_j}} + \beta_j \; \forall\, j$$

Question: how to perform training after plugging in the BN operations?

$$\min_{\boldsymbol{W}} \; \frac{1}{m} \sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \sigma\left(\boldsymbol{W}_k \mathrm{BN}\left(\sigma\left(\boldsymbol{W}_{k-1} \ldots \mathrm{BN}\left(\sigma\left(\boldsymbol{W}_1 \boldsymbol{x}_i\right)\right)\right)\right)\right)\right) + \Omega\left(\boldsymbol{W}\right)$$

Answer: for all j, $\mathrm{BN}_{\gamma_j,\boldsymbol{\beta}_j}\left(\boldsymbol{Z}_j\right)$ is nothing but a differentiable function of $\boldsymbol{Z}_j$, γ_j, and β_j — chain rule applies!

- $\mu_{\boldsymbol{Z}_j}$ and $\sigma_{\boldsymbol{Z}_j}$ are differentiable functions of $\boldsymbol{Z}_j$, and $\left(\boldsymbol{Z}_j, \gamma_j, \beta_j\right) \mapsto \mathrm{BN}_{\gamma_j,\boldsymbol{\beta}_j}\left(\boldsymbol{Z}_j\right)$ is a vector-to-vector mapping
- Any col $\boldsymbol{Z}_j$ depends on all $\boldsymbol{x}_k$'s in the current mini-batch B as $\boldsymbol{z}_i \longleftarrow \boldsymbol{x}_i$ for $i = 1, \ldots, |B|$
- Without BN: $\nabla_{\boldsymbol{W}} \frac{1}{|B|} \sum_{k=1}^{|B|} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right) = \frac{1}{|B|} \sum_{k=1}^{|B|} \nabla_{\boldsymbol{W}} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right)$, the summands can be computed in parallel and then aggregated
  With BN: $\nabla_{\boldsymbol{W}} \frac{1}{|B|} \sum_{k=1}^{|B|} \ell\left(\boldsymbol{W}; \boldsymbol{x}_k, \boldsymbol{y}_k\right)$ has to be computed altogether, due to the inter-dependency across the summands

Batch normalization: implementation details

$$\mathrm{BN}_{\gamma_j,\boldsymbol{\beta}_j}\left(\boldsymbol{Z}_j\right) = \gamma_j \frac{\boldsymbol{Z}_j - \mu_{\boldsymbol{Z}_j}}{\sigma_{\boldsymbol{Z}_j}} + \beta_j \ \forall\ j$$

What about validation/test, where only a single sample is seen each time?

idea: use the average $\mu_{\boldsymbol{z}^j}$'s and $\sigma_{\boldsymbol{z}^j}$'s over the training data (γ_j's and β_j's are learned)

In practice, collect the momentum-weighted running averages: e.g., for each hidden node with BN,

$$\overline{\mu} = (1-\eta)\,\overline{\mu}_{old} + \eta\mu_{new}$$
$$\overline{\sigma} = (1-\eta)\,\overline{\sigma}_{old} + \eta\sigma_{new}$$

with e.g., $\eta = 0.1$. In PyTorch, `torch.nn.BatchNorm1d`, `torch.nn.BatchNorm2d`, `torch.nn.BatchNorm3d` depending on the input shapes

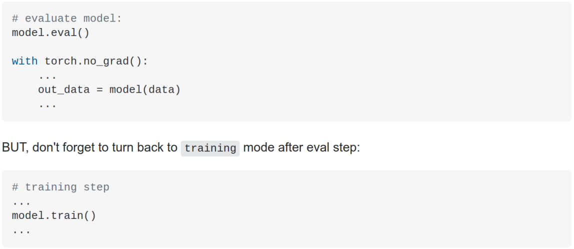
Training and evaluation modes

In practice, collect the momentum-weighted running averages: e.g., for each hidden node with BN,

$$\overline{\mu} = (1-\eta)\,\overline{\mu}_{old} + \eta\mu_{new}$$
$$\overline{\sigma} = (1-\eta)\,\overline{\sigma}_{old} + \eta\sigma_{new}$$

with e.g., $\eta = 0.1$.

- Different behaviors in **training** and **evaluation** modes for BatchNorm (similarly for Dropout discussed later)
- Pytorch implements `.train()` and `.eval()` to switch between the modes



```
# evaluate model:
model.eval()

with torch.no_grad():
    ...
    out_data = model(data)
    ...
```

BUT, don't forget to turn back to `training` mode after eval step:

```
# training step
...
model.train()
...
```

Batch normalization: implementation details

Question: BN before or after the activation?

$$\boldsymbol{W}_2\sigma(\boldsymbol{W}_1\boldsymbol{x}_i) \longrightarrow \boldsymbol{W}_2\mathrm{BN}(\sigma(\boldsymbol{W}_1\boldsymbol{x}_i)) \quad \text{(after)}$$
$$\boldsymbol{W}_2\sigma(\boldsymbol{W}_1\boldsymbol{x}_i) \longrightarrow \boldsymbol{W}_2(\sigma(\mathrm{BN}(\boldsymbol{W}_1\boldsymbol{x}_i))) \quad \text{(before)}$$

- The original paper [Ioffe and Szegedy, 2015] proposed the "before" version (most of the original intuition has since proved wrong)
- But the "after" version is more intuitive as we have seen
- Both are used in practice and debatable which one is more effective
  * https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/
  * https://blog.paperspace.com/busting-the-myths-about-batch-normalization/
  * https://github.com/gcr/torch-residual-networks/issues/5
  * [Chen et al., 2019]

Why BN works?

Short answer: we don't know exactly

Long answer:

- Originally proposed to deal with *internal covariate shift* [Ioffe and Szegedy, 2015]
- The original intuition later proved wrong and BN is shown to make the optimization problem "nicer" (or "smoother") [Santurkar et al., 2018, Lipton and Steinhardt, 2019]
- Yet another explanation from optimization perspective [Kohler et al., 2019]
- A good research topic

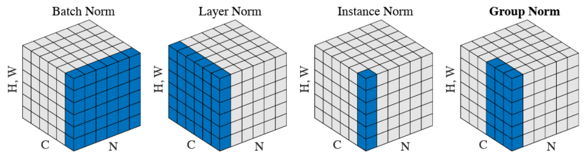
Batch PCA/whitening?

fixing the feature scaling makes the landscape "nicer"—derivatives and curvatures in all directions are roughly even in magnitudes. So for DNNs,

- Add (pre-)processing to input data
  * zero-center
  * normalization
  * PCA or whitening (less common)
- Add batch-processing steps to some/all hidden layers
  * Batch normalization
  * Batch PCA or whitening? Doable but requires a lot of work [Huangi et al., 2018, Huang et al., 2019, Wang et al., 2019]

normalization is most popular due to the simplicity

Zoo of normalization



Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Credit: [Wu and He, 2018]

normalization in different directions/groups of the data tensors

- N is the batch axis
- C is the # output nodes (often called "channels" in CNN context)
- WH is the per output dimension (1 for fully connected, but 2D for CNNs)

layer/group normalization:

- small N (batch size) is possible
- simplicity: training/test normalizations are consistent

Zoo of normalization

weight normalization: decompose the weight as magnitude and direction $\boldsymbol{w} = g \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$ and perform optimization in $(g, \boldsymbol{v})$ space

An Overview of Normalization Methods in Deep Learning
`https://mlexplained.com/2018/11/30/`
`an-overview-of-normalization-methods-in-deep-learning/`

Check out PyTorch normalization layers

`https://pytorch.org/docs/stable/nn.html#normalization-layers`

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Regularization to avoid overfitting

Training DNNs $\min_{\boldsymbol{W}} \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \lambda\Omega\left(\boldsymbol{W}\right)$ with **explicit regularization** Ω. But which Ω?

- $\Omega\left(\boldsymbol{W}\right) = \sum_k \|\boldsymbol{W}_k\|_F^2$ where k indexes the layers — penalizes large values in $\boldsymbol{W}$ and hence avoids steep changes (set `weight_decay` as λ in `torch.optim.xxxx`)
- $\Omega\left(\boldsymbol{W}\right) = \sum_k \|\boldsymbol{W}_k\|_1$ — promotes sparse $\boldsymbol{W}_k$'s (i.e., many entries in $\boldsymbol{W}_k$'s to be near zero; good for feature selection)

```
 l1_reg = torch.zeros(1)
for W in model.parameters():
  l1_reg += W.norm(1)
```

- $\Omega\left(\boldsymbol{W}\right) = \|\boldsymbol{J}_{\mathrm{DNN}_{\boldsymbol{W}}}\left(\boldsymbol{x}\right)\|_F^2$ — promotes slow change of the function represented by $\mathrm{DNN}_{\boldsymbol{W}}$ [Varga et al., 2017, Hoffman et al., 2019, Chan et al., 2019]
- Constraints, $\delta_C\left(\boldsymbol{W}\right) \doteq \begin{cases} 0 & \boldsymbol{W} \in C \\ \infty & \boldsymbol{W} \notin C \end{cases}$, e.g., binary, norm bound
- many others!

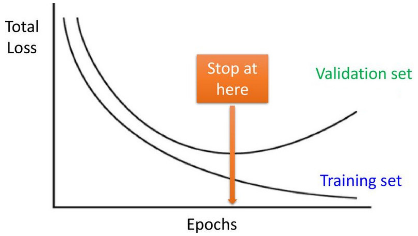
Implicit regularization

Training DNNs $\min_{\boldsymbol{W}} \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathrm{DNN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right) + \lambda\Omega\left(\boldsymbol{W}\right)$ with **implicit regularization —** operation that is not built into the objective but avoids overfitting

- early stopping
- (batch) normalization
- dropout
- algorithm choice
- etc

Early stopping

A practical/pragmatic stopping strategy: **early stopping**

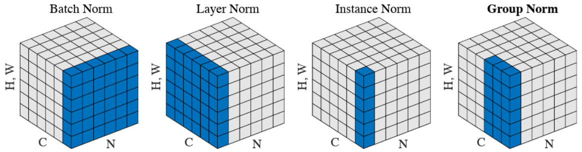


... periodically check the validation error and stop when it doesn't improve

Intuition: avoid the model to be too specialized/perfect for the training data

More concrete math examples: [Bishop, 1995, Sjöberg and Ljung, 1995]

Batch/general normalization



Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Credit: [Wu and He, 2018]

normalization in different directions/groups of the data tensors

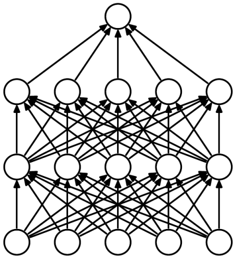
weight normalization: decompose the weight as magnitude and direction $\boldsymbol{w} = g \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$ and perform optimization in $(g, \boldsymbol{v})$ space

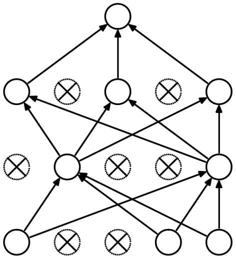
An Overview of Normalization Methods in Deep Learning

https://mlexplained.com/2018/11/30/

an-overview-of-normalization-methods-in-deep-learning/

Dropout


(a) Standard Neural Net

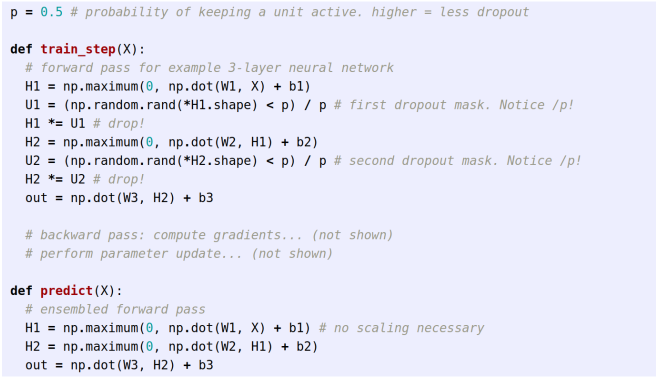

(b) After applying dropout.

Credit: [Srivastava et al., 2014]

Idea: kill each non-output neuron with probability $1-p$, called **Dropout**

- perform Dropout independently for each training sample and each iteration
- for each neuron, if the original output is x, then the expected output with Dropout: px. So rescale the actual output by $1/p$
- no Dropout at test time!

Dropout: implementation details



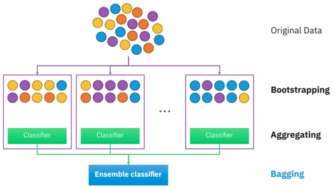
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Credit: Stanford CS231N

What about derivatives? Back-propagation for each sample and then aggregate

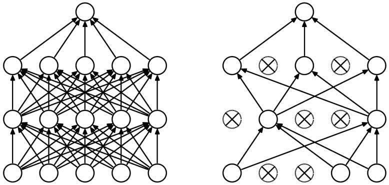
PyTorch: `torch.nn.Dropout`, `torch.nn.Dropout2d`, `torch.nn.Dropout3d`

Why Dropout?



Credit: Wikipedia

bagging can avoid overfitting


(a) Standard Neural Net (b) After applying dropout.

Credit: [Srivastava et al., 2014]

For an n-node network, $O(2^n)$ possible sub-networks.

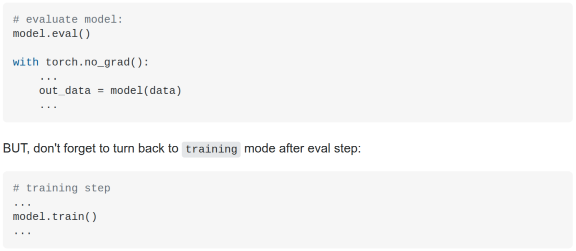
Consider the average/ensemble prediction $\mathbb{E}_{\text{SN}}\left[\text{SN}\left(\boldsymbol{x}\right)\right]$ over 2^n of sub-networks and the new objective

$$F\left(\boldsymbol{W}\right) \doteq \frac{1}{m}\sum_{i=1}^{m} \ell\left(\boldsymbol{y}_i, \mathbb{E}_{\text{SN}}\left[\text{SN}_{\boldsymbol{W}}\left(\boldsymbol{x}_i\right)\right]\right)$$

Mini-batch SGD with Dropout samples data point and model simultaneously (stochastic composite optimization [Wang et al., 2016, Wang et al., 2017])

Implementation details

- Different behaviors in **training** and **evaluation** modes for Dropout (similarly for BatchNorm discussed earlier)
- Pytorch implements `.train()` and `.eval()` to switch between the modes



```
# evaluate model:
model.eval()

with torch.no_grad():
    ...
    out_data = model(data)
    ...
```

BUT, don't forget to turn back to `training` mode after eval step:

```
# training step
...
model.train()
...
```

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

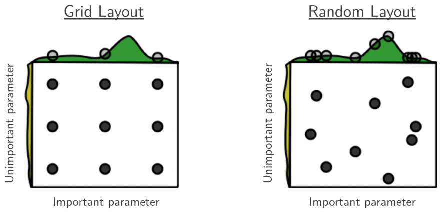
Suggested reading

Hyperparameter search

...tunable parameters (vs. learnable parameters, or optimization variables)

- Network architecture (depth, width, activation, loss, etc)
- Optimization methods
- Initialization schemes
- Initial LR and LR schedule/parameters
- regularization methods and parameters
- etc

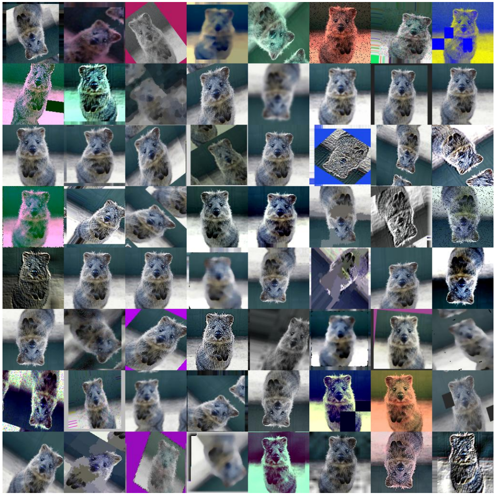
https://cs231n.github.io/neural-networks-3/#hyper



Credit: [Bergstra and Bengio, 2012]

Data augmentation

- More relevant data always help!
- Fetch more external data
- Generate more internal data: generate based on whatever you want to be robust to
    * vision: translation, rotation, background, noise, deformation, flipping, blurring, occlusion, etc



Credit: https://github.com/aleju/imgaug

See one example here https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Outline

Three design choices

Training algorithms

- Which method
- Where to start
- When to stop

Tricks

- Data Normalization
- Regularization
- Hyperparameter search, data augmentation

Suggested reading

Suggested reading

- Chap 7, Deep Learning (Goodfellow et al)
- Sun, Ruoyu. "Optimization for deep learning: theory and algorithms." arXiv preprint arXiv:1912.08957 (2019).
- UIUC IE598-ODL Optimization Theory for Deep Learning https://wiki.illinois.edu/wiki/display/IE598ODLSP19/IE598-ODL++Optimization+Theory+for+Deep+Learning
- Stanford CS231n course notes: Neural Networks Part 1: Setting up the Architecture https://cs231n.github.io/neural-networks-1/
- Stanford CS231n course notes: Neural Networks Part 2: Setting up the Data and the Loss https://cs231n.github.io/neural-networks-2/
- Stanford CS231n course notes: Neural Networks Part 3: Learning and Evaluation https://cs231n.github.io/neural-networks-3/
- http://neuralnetworksanddeeplearning.com/chap3.html

References i

[Anil et al., 2020] Anil, R., Gupta, V., Koren, T., Regan, K., and Singer, Y. (2020). **Second order optimization made practical.** *arXiv:2002.09018*.

[Arjovsky et al., 2016] Arjovsky, M., Shah, A., and Bengio, Y. (2016). **Unitary evolution recurrent neural networks.** In *International Conference on Machine Learning*, pages 1120–1128.

[Bansal et al., 2018] Bansal, N., Chen, X., and Wang, Z. (2018). **Can we gain more from orthogonality regularizations in training deep cnns?** In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 4266–4276. Curran Associates Inc.

[Baskerville et al., 2020] Baskerville, N. P., Keating, J. P., Mezzadri, F., and Najnudel, J. (2020). **The loss surfaces of neural networks with general activation functions.** *arXiv:2004.03959*.

[Baydin et al., 2017] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2017). **Automatic differentiation in machine learning: a survey.** *The Journal of Machine Learning Research*, 18(1):5595–5637.

References ii

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). **Random search for hyper-parameter optimization.** *Journal of machine learning research*, 13(Feb):281–305.

[Bishop, 1995] Bishop, C. M. (1995). **Regularization and complexity control in feed-forward networks.** In *International Conference on Artificial Neural Networks ICANN*.

[Bottou and Bousquet, 2008] Bottou, L. and Bousquet, O. (2008). **The tradeoffs of large scale learning.** In *Advances in neural information processing systems*, pages 161–168.

[Byrd et al., 2016] Byrd, R. H., Hansen, S. L., Nocedal, J., and Singer, Y. (2016). **A stochastic quasi-newton method for large-scale optimization.** *SIAM Journal on Optimization*, 26(2):1008–1031.

[Chan et al., 2019] Chan, A., Tay, Y., Ong, Y. S., and Fu, J. (2019). **Jacobian adversarially regularized networks for robustness.** *arXiv:1912.10185*.

[Chauhan et al., 2018] Chauhan, V. K., Sharma, A., and Dahiya, K. (2018). **Stochastic trust region inexact newton method for large-scale machine learning.** *arXiv:1812.10426*.

References iii

[Chen et al., 2019] Chen, G., Chen, P., Shi, Y., Hsieh, C.-Y., Liao, B., and Zhang, S. (2019). **Rethinking the usage of batch normalization and dropout in the training of deep neural networks.** *arXiv:1905.05928*.

[Choromanska et al., 2015] Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. (2015). **The loss surfaces of multilayer networks.** In *Artificial intelligence and statistics*, pages 192–204.

[Chu et al., 2026] Chu, Y.-C., Boukas, A., and Udell, M. (2026). **Snarenet: Flexible repair layers for neural networks with hard constraints.** *arXiv preprint arXiv:2602.09317*.

[Curtis and Shi, 2019] Curtis, F. E. and Shi, R. (2019). **A fully stochastic second-order trust region method.** *arXiv:1911.06920*.

[Dauphin et al., 2014] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). **Identifying and attacking the saddle point problem in high-dimensional non-convex optimization.** In *Advances in neural information processing systems*, pages 2933–2941.

References iv

[Defazio and Bottou, 2019] Defazio, A. and Bottou, L. (2019). **Scaling laws for the principled design, initialization and preconditioning of relu networks.** *arXiv:1906.04267*.

[Hoffman et al., 2019] Hoffman, J., Roberts, D. A., and Yaida, S. (2019). **Robust learning with jacobian regularization.** *arXiv:1908.02729*.

[Huang et al., 2019] Huang, L., Zhou, Y., Zhu, F., Liu, L., and Shao, L. (2019). **Iterative normalization: Beyond standardization towards efficient whitening.** pages 4869–4878. IEEE.

[Huangi et al., 2018] Huangi, L., Huangi, L., Yang, D., Lang, B., and Deng, J. (2018). **Decorrelated batch normalization.** pages 791–800. IEEE.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). **Batch normalization: Accelerating deep network training by reducing internal covariate shift.** In *The 32nd International Conference on Machine Learning*.

References v

[Kohler et al., 2019] Kohler, J. M., Daneshmand, H., Lucchi, A., Hofmann, T., Zhou, M., and Neymeyr, K. (2019). **Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization.** In *The 22nd International Conference on Artificial Intelligence and Statistics*.

[Lezcano-Casado and Martínez-Rubio, 2019] Lezcano-Casado, M. and Martínez-Rubio, D. (2019). **Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group.** *arXiv1901.08428*.

[Li et al., 2020] Li, J., Fuxin, L., and Todorovic, S. (2020). **Efficient riemannian optimization on the stiefel manifold via the cayley transform.** *arXiv:2002.01113*.

[Lipton and Steinhardt, 2019] Lipton, Z. C. and Steinhardt, J. (2019). **Troubling trends in machine learning scholarship.** *ACM Queue*, 17(1):80.

[Martens and Grosse, 2015] Martens, J. and Grosse, R. (2015). **Optimizing neural networks with kronecker-factored approximate curvature.** In *International conference on machine learning*, pages 2408–2417.

References vi

[Min and Azizan, 2024] Min, Y. and Azizan, N. (2024). **Hardnet: Hard-constrained neural networks with universal approximation guarantees.** *arXiv preprint arXiv:2410.10807*.

[Pascanu et al., 2014] Pascanu, R., Dauphin, Y. N., Ganguli, S., and Bengio, Y. (2014). **On the saddle point problem for non-convex optimization.** *arXiv preprint arXiv:1405.4604*.

[Roosta-Khorasani and Mahoney, 2018] Roosta-Khorasani, F. and Mahoney, M. W. (2018). **Sub-sampled newton methods.** *Mathematical Programming*, 174(1-2):293–326.

[Santurkar et al., 2018] Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2018). **How does batch normalization help optimization?** In *Advances in Neural Information Processing Systems*, pages 2483–2493.

[Sjöberg and Ljung, 1995] Sjöberg, J. and Ljung, L. (1995). **Overtraining, regularization and searching for a minimum, with application to neural networks.** *International Journal of Control*, 62(6):1391–1407.

References vii

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). **Dropout: a simple way to prevent neural networks from overfitting.** *The journal of machine learning research*, 15(1):1929–1958.

[Staib et al., 2020] Staib, M., Reddi, S. J., Kale, S., Kumar, S., and Sra, S. (2020). **Escaping saddle points with adaptive gradient methods.** *arXiv:1901.09149*.

[Sun, 2019] Sun, R. (2019). **Optimization for deep learning: theory and algorithms.** *arXiv:1912.08957*.

[Varga et al., 2017] Varga, D., Csiszárik, A., and Zombori, Z. (2017). **Gradient regularization improves accuracy of discriminative models.** *arXiv:1712.09936*.

[Wang et al., 2016] Wang, M., Fang, E. X., and Liu, H. (2016). **Stochastic compositional gradient descent: algorithms for minimizing compositions of expected-value functions.** *Mathematical Programming*, 161(1-2):419–449.

[Wang et al., 2017] Wang, M., Liu, J., and Fang, E. X. (2017). **Accelerating stochastic composition optimization.** *The Journal of Machine Learning Research*, 18(1):3721–3743.

References viii

[Wang et al., 2019] Wang, W., Dang, Z., Hu, Y., Fua, P., and Salzmann, M. (2019). **Backpropagation-friendly eigendecomposition.** In *Advances in Neural Information Processing Systems*, pages 3156–3164.

[Wu and He, 2018] Wu, Y. and He, K. (2018). **Group normalization.** In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19.