Optimization

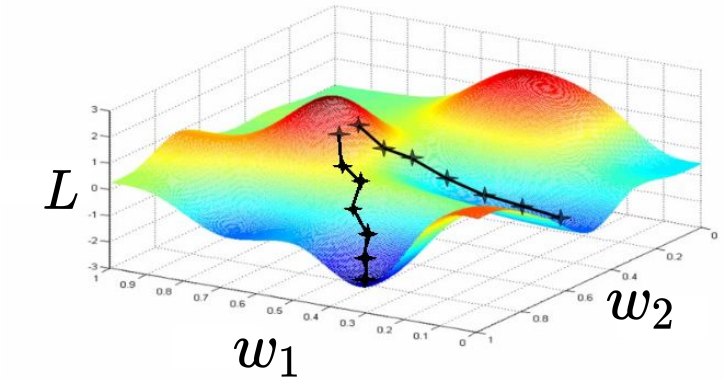
Today

- Today is about optimizing a deep model using a training set
- Remember from last time:
  - A model that computes an output using parameters (**W**,**b**)
  - A loss *L* between the output and the desired outputs (e.g. labels)
  - A method for computing the gradient **g** of *L* w.r.t. the parameters

Gradient descent

- Last time you tried *minibatch GD*
  - Random subsets of training data in each iteration
- But why not the full set, given the loss function we are trying to minimize?
  - MSE loss: $L(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} \|y_i - \hat{y}_i)\|^2$
  - Log loss: $L(\hat{y}, y) = -\frac{1}{m} \sum_{i=1}^{m} y_i \log \hat{y}_i$
- One loss function → one big gradient → one step at a time?

https://infinitygc.com.au/wp-content/uploads/2018/03/88007-02.jpg

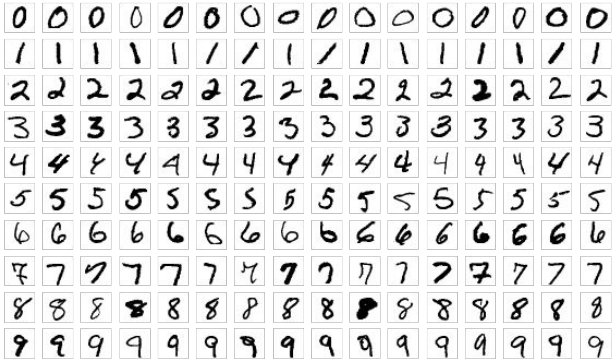


http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Batch gradient descent

- Infeasible (impossible) for huge training data sets
  - Just think of the forward pass in a small 3-layer network:
- If possible, overall slow
  - Huge efforts for computing only a single, average gradient
- Possible redundancy
  - Training sets often contain many near-duplicate examples
  - Using all examples means a waste of resources
- Determinism
  - May not be preferred
  - We only have an incomplete sample of data

$$\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{h}_2 = f(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$$
$$\hat{\mathbf{y}} = f(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)$$



https://en.wikipedia.org/wiki/MNIST_database#/media/File:MnistExamples.png

Additional challenges

- Non-convexity of *L*
  - Local minima
  - Saddle points
- Steep cliffs
  - Exploding gradients
- The *empirical risk*
  - We may only learn from a training set
    - But we really care about the future performance on a test set
  - We need to use a smooth (differentiable) loss (log loss)
    - But we really care about the (non-differentiable) 0-1 classification loss

Variants of stochastic gradient descent (SGD)

- “Pure” SGD (or online SGD)
  - Select one random training example at a time, compute gradient, update parameters
  - High variance, but many parameter updates
- Minibatch SGD
  - Select a random subset of training examples, compute gradient, update parameters
  - Trade-off between
    - exploration/variance (small batch size),
    - speed (small batch size),
    - stability in gradient estimates (large batch size)

Minibatch SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

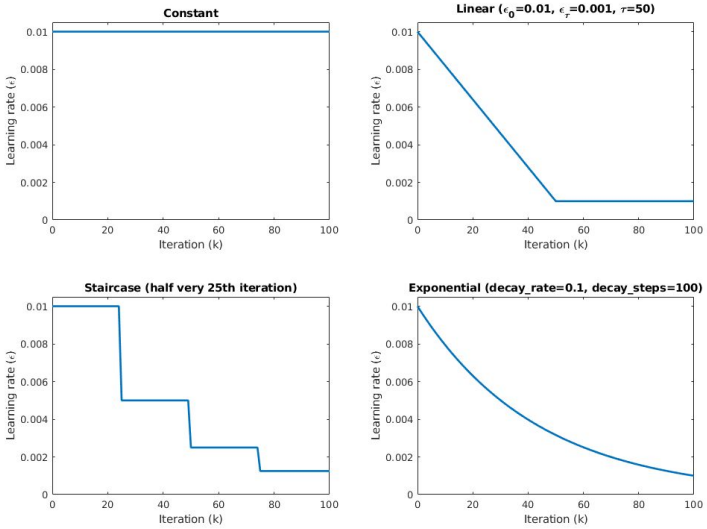
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

end while

Learning rate schedule

- Common approaches for SGD
  - Constant: $\epsilon_k = \epsilon_0$
  - Linear decay: $\epsilon_k = (1 - \frac{k}{\tau})\epsilon_0 + \frac{k}{\tau}\epsilon_\tau$
  - Piecewise constant or “staircase”
  - Exponential: $\epsilon_k = \epsilon_0 \cdot \text{decay\_rate}^{k/\text{decay\_steps}}$
- Resources
  - https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate
  - A very common approach is the `StepLR` schedule

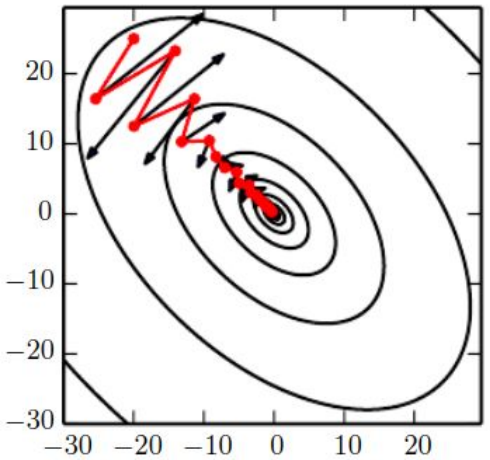


Momentum

- Here's the update rule for (S)GD: $\mathbf{W} \leftarrow \mathbf{W} - \epsilon \mathbf{g}$
- Momentum tries to accelerate learning by including previous gradients
- Define a velocity vector **v**, initialized to zero
- The momentum update rule:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$$
$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

Momentum parameter



Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
end while

Nesterov momentum

- Instead of directly updating the velocity using the current gradient, *predict* a new gradient:
  $\tilde{\mathbf{W}} \leftarrow \mathbf{W} + \alpha \mathbf{v}$
  estimate $\hat{\mathbf{y}}$ using $\tilde{\mathbf{W}}$ as model parameters...
  $\mathbf{g} \leftarrow \nabla L(\hat{\mathbf{y}}, \mathbf{y})$
- After then, it's standard momentum:
  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$
  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$

Nesterov momentum

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.

Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Adaptive learning rates

- The learning rate is a *hyperparameter*, which is hard to tune
- In a high-dimensional error landscape, it could be helpful to:
  - Decrease the step in steep directions
  - Increase the step in almost flat directions

Adaptive learning rates

- Three variants in the book
  - AdaGrad
  - RMSProp
  - Adam
- And many more...

AdaGrad

- Divide current gradient by magnitude of entire past history of gradients
- Rapid decay in learning rate

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.
Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.
end while

RMSProp

- Add a momentum-like exponential decay to the gradient history
- Gradients in extreme past have less influence
- Less rapid decay in learning rate

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

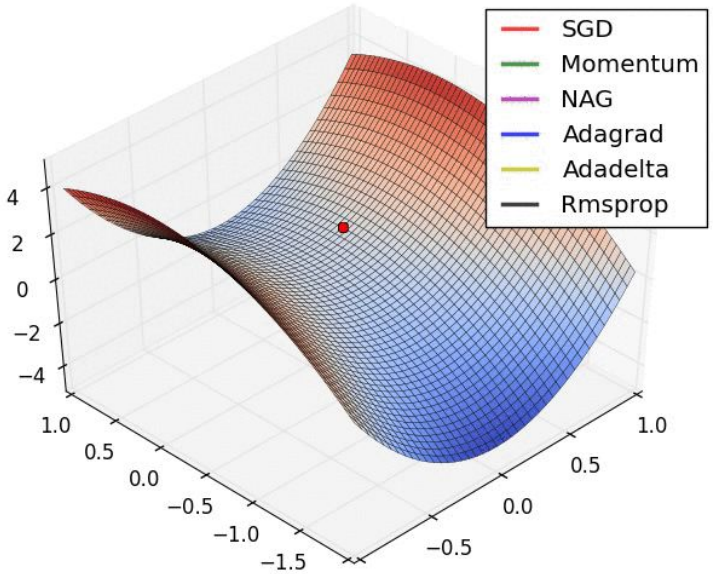
Adam

- Call the 1st and 2nd power of the gradient *moments*
- Adam uses a momentum-like decay on both moments
- In addition, it uses *bias correction* to avoid initial instabilities of the moments

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Adaptive learning rates



http://cs231n.github.io/neural-networks-3/

Batch normalization

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe SIOFFE@GOOGLE.COM
Christian Szegedy SZEGEDY@GOOGLE.COM
Google, 1600 Amphitheatre Pkwy, Mountain View, CA 94043

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs.

minimize the loss

$$\Theta = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \ell(\mathrm{x}_i, \Theta)$$

where $\mathrm{x}_{1\ldots N}$ is the training data set. With SGD, the training proceeds in steps, at each step considering a *mini-batch* $\mathrm{x}_{1\ldots m}$ of size m. Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch $\frac{1}{m}\sum_{i=1}^{m} \frac{\partial \ell(\mathrm{x}_i,\Theta)}{\partial \Theta}$ is an estimate of the gradient over the

BatchNorm

- Recall the output (the neuronal activity) at any layer *i*:
  $\mathbf{h}_i = \phi(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i) = \phi(\mathbf{a}_i)$

Activation function (e.g. ReLu, sigmoid)

- If we are using a (mini)batch of *m* vectors, we get a data matrix **H** instead:
  $\mathbf{H}_i = \phi(\mathbf{W}_i \mathbf{H}_{i-1} + \mathbf{b}_i)$
- BatchNorm (or BN) normalizes the *i*'th layer using the *m* samples:
  $\mathbf{H}'_i = \frac{\mathbf{H}_i - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}$

A column vector with the means of all rows of **H**

A column vector with the standard deviations of all rows of **H**

BatchNorm $\mathbf{H}'_i = \frac{\mathbf{H}_i - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}$

- The computations of mean/std:
  $\boldsymbol{\mu}_i = \frac{1}{m}\sum_{j=1}^{m} \mathbf{H}_{i,j} \qquad \boldsymbol{\sigma}_i = \sqrt{\delta + \frac{1}{m}\sum_{j=1}^{m}(\mathbf{H}_{i,j} - \boldsymbol{\mu}_{i,j})^2}$
  are part of the feedforward chain (you can see them as a pseudo-layer)
- This means that backprop will “see” these operations and take them into account when computing gradients
- This in return means that learning can focus on other stuff than trying to bring the activations to a “good” range

BatchNorm $\mathbf{H}_i' = \frac{\mathbf{H}_i - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i}$

- All units in the network are now normalized - this stabilizes training
  - Read the section in the book for more in-depth of this
- The final stage of BN scales/shifts the layer responses by two trainable parameters:

  $\mathbf{H}_i' \leftarrow \boldsymbol{\gamma}_i \mathbf{H}_i' + \boldsymbol{\beta}_i$
- During *evaluation/test*, these learned parameters are fixed
  - For the mean/std, running averages are simply accumulated during training

BatchNorm

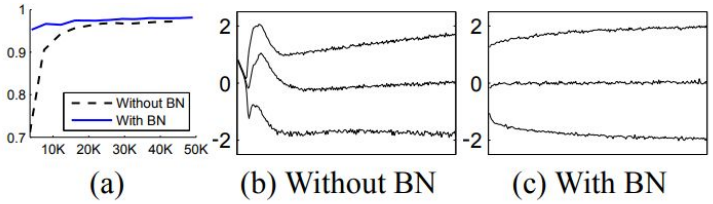


Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

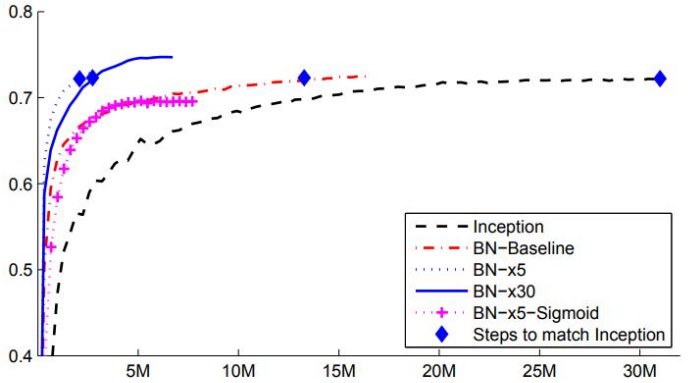


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Ioffe & Szegedy, 2015

Challenge

- Use solution from lecture 3 (MLP) as a starting point
  - Remove the normalization to [-0.5, 0.5] pixel values in the Cifar10 class, i.e. use the full pixel range [0, 255] - remember to still convert to float32
  - Use 1 hidden layer with 100 neurons
  - Now train on the unnormalized pixels without and with BatchNorm (applied after ReLU)
    - When using BatchNorm, use model.train()/model.eval() when training/validating (because the layer behaves differently under training and inference)
  - Try a learning rate schedule from torch.optim.lr_scheduler
  - Try replacing the SGD optimizer with the Adam optimizer from torch.optim
- Bonus:
  - Put back the [-0.5, 0.5] pixel normalization
  - Try to do batch GD instead of minibatch SGD
    It's expensive to calculate the gradient, but the gradient has less variance so you can increase the learning rate. Is it enough to compensate?