Machine Learning – Lecture 12

Tricks of the Trade

03.12.2018

Bastian Leibe
RWTH Aachen
http://www.vision.rwth-aachen.de
leibe@vision.rwth-aachen.de

Course Outline

• Fundamentals
  ▶ Bayes Decision Theory
  ▶ Probability Density Estimation

• Classification Approaches
  ▶ Linear Discriminants
  ▶ Support Vector Machines
  ▶ Ensemble Methods & Boosting
  ▶ Random Forests

• Deep Learning
  ▶ Foundations
  ▶ Convolutional Neural Networks
  ▶ Recurrent Neural Networks

Topics of This Lecture

• Recap: Optimization
  ▶ Effect of optimizers

• Tricks of the Trade
  ▶ Shuffling
  ▶ Data Augmentation
  ▶ Normalization

• Nonlinearities

• Initialization

• Advanced techniques
  ▶ Batch Normalization
  ▶ Dropout

Recap: Computational Graphs

Recap: Automatic Differentiation

Recap: Choosing the Right Learning Rate

Course Outline

• Fundamentals
  ▶ Bayes Decision Theory
  ▶ Probability Density Estimation

• Classification Approaches
  ▶ Linear Discriminants
  ▶ Support Vector Machines
  ▶ Ensemble Methods & Boosting
  ▶ Random Forests

• Deep Learning
  ▶ Foundations
  ▶ Convolutional Neural Networks
  ▶ Recurrent Neural Networks

Topics of This Lecture

• Recap: Optimization
  ▶ Effect of optimizers

• Tricks of the Trade
  ▶ Shuffling
  ▶ Data Augmentation
  ▶ Normalization

• Nonlinearities

• Initialization

• Advanced techniques
  ▶ Batch Normalization
  ▶ Dropout

Recap: Computational Graphs

Recap: Automatic Differentiation

Recap: Choosing the Right Learning Rate

Course Outline

• Fundamentals
  ▶ Bayes Decision Theory
  ▶ Probability Density Estimation

• Classification Approaches
  ▶ Linear Discriminants
  ▶ Support Vector Machines
  ▶ Ensemble Methods & Boosting
  ▶ Random Forests

• Deep Learning
  ▶ Foundations
  ▶ Convolutional Neural Networks
  ▶ Recurrent Neural Networks

Topics of This Lecture

• Recap: Optimization
  ▶ Effect of optimizers

• Tricks of the Trade
  ▶ Shuffling
  ▶ Data Augmentation
  ▶ Normalization

• Nonlinearities

• Initialization

• Advanced techniques
  ▶ Batch Normalization
  ▶ Dropout

Recap: Computational Graphs

Recap: Automatic Differentiation

Recap: Choosing the Right Learning Rate

Course Outline

• Fundamentals
  ▶ Bayes Decision Theory
  ▶ Probability Density Estimation

• Classification Approaches
  ▶ Linear Discriminants
  ▶ Support Vector Machines
  ▶ Ensemble Methods & Boosting
  ▶ Random Forests

• Deep Learning
  ▶ Foundations
  ▶ Convolutional Neural Networks
  ▶ Recurrent Neural Networks

Topics of This Lecture

• Recap: Optimization
  ▶ Effect of optimizers

• Tricks of the Trade
  ▶ Shuffling
  ▶ Data Augmentation
  ▶ Normalization

• Nonlinearities

• Initialization

• Advanced techniques
  ▶ Batch Normalization
  ▶ Dropout

Recap: Computational Graphs

Recap: Automatic Differentiation

Recap: Choosing the Right Learning Rate
Separate, Adaptive Learning Rates

- **Problem**
  - In multilayer nets, the appropriate learning rates can vary widely between weights.
  - The **magnitudes of the gradients** are often very different for the different layers, especially if the initial weights are small.
  - Gradients can get very small in the early layers of deep nets.
  - The **fan-in** of a unit determines the size of the “overshoot” effect when changing multiple weights simultaneously to correct the same error.
    - The fan-in often varies widely between layers.

- **Solution**
  - Use a global learning rate, multiplied by a local gain per weight (determined empirically).

Better Adaptation: RMSProp

- **Motivation**
  - The magnitude of the gradient can be very different for different weights and can change during learning.
  - This makes it hard to choose a single global learning rate.
  - For batch learning, we can deal with this by only using the sign of the gradient, but we need to generalize this for minibatches.

- **Idea of RMSProp**
  - Divide the gradient by a running average of its recent magnitude
    \[ \text{MeanSq}(w_{ij}, t) = 0.9 \text{MeanSq}(w_{ij}, t-1) + 0.1 \left( \frac{\partial E}{\partial w_{ij}}(t) \right)^2 \]
  - Divide the gradient by \( \text{sqrt}(\text{MeanSq}(w_{ij}, t)) \).

Other Optimizers

- AdaGrad [Duchi '10]
- AdaDelta [Zeiler '12]
- Adam [Ba & Kingma '14]

- **Notes**
  - All of those methods have the goal to make the optimization less sensitive to parameter settings.
  - Adam is currently becoming the quasi-standard.
Trick: Patience

- Saddle points dominate in high-dimensional spaces!

Learning often doesn’t get stuck, you just may have to wait...

Reducing the Learning Rate

- Final improvement step after convergence is reached
  - Reduce learning rate by a factor of 10.
  - Continue training for a few epochs.
  - Do this 1-3 times, then stop training.

- Effect
  - Turning down the learning rate will reduce the random fluctuations in the error due to different gradients on different minibatches.

  - Be careful: Do not turn down the learning rate too soon!
    - Further progress will be much slower/impossible after that.

Summary

- Deep multi-layer networks are very powerful.

- But training them is hard!
  - Complex, non-convex learning problem
  - Local optimization with stochastic gradient descent

- Main issue: getting good gradient updates for the lower layers of the network
  - Many seemingly small details matter!
  - Weight initialization, normalization, data augmentation, choice of nonlinearities, choice of learning rate, choice of optimizer,…

  - In the following, we will take a look at the most important factors

Topics of This Lecture

- Recap: Optimization
  - Effect of optimizers

- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

- Nonlinearities

- Initialization

- Advanced techniques
  - Batch Normalization
  - Dropout

Shuffling the Examples

- Ideas
  - Networks learn fastest from the most unexpected sample.

  - It is advisable to choose a sample at each iteration that is most unfamiliar to the system.
    - E.g. a sample from a different class than the previous one.
    - This means, do not present all samples of class A, then all of class B.

  - A large relative error indicates that an input has not been learned by the network yet, so it contains a lot of information.

  - It can make sense to present such inputs more frequently.

  - But: be careful, this can be disastrous when the data are outliers.

- Practical advice
  - When working with stochastic gradient descent or minibatches, make use of shuffling.

Data Augmentation

- Idea
  - Augment original data with synthetic variations to reduce overfitting

- Example augmentations for images
  - Cropping
  - Zooming
  - Flipping
  - Color PCA
Data Augmentation

- **Effect**
  - Much larger training set
  - Robustness against expected variations

- **During testing**
  - When cropping was used during training, need to again apply crops to get same image size.
  - Beneficial to also apply flipping during test.
  - Applying several ColorPCA variations can bring another ~1% improvement, but at a significantly increased runtime.

Augmented training data (from one original image)

Practical Advice

**APPLY ALL THE AUGMENTATIONS**

Normalization

- **Motivation**
  - Consider the Gradient Descent update steps
    \[ w^{(r+1)}_{kj} = w^{(r)}_{kj} - \eta \frac{\partial E(w)}{\partial w_{kj}} |_{w^{(r)}} \]
  - From backpropagation, we know that
    \[ \frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j} \]
  - When all of the components of the input vector \( y \) are positive, all of the updates of weights that feed into a node will be of the same sign.
  - Weights can only all increase or decrease together.
  - Slow convergence

Normalized the Inputs

- **Convergence is fastest if**
  - The mean of each input variable over the training set is zero.
  - The inputs are scaled such that all have the same covariance.
  - Input variables are uncorrelated if possible.

- **Advisable normalization steps** *(for MLPs only, not for CNNs)*
  - Normalize all inputs that an input unit sees to zero-mean, unit covariance.
  - If possible, try to decorrelate them using PCA (also known as Karhunen-Loève expansion).

Choosing the Right Sigmoid

- **Normalization is also important for intermediate layers**
  - Symmetric sigmoids, such as tanh, often converge faster than the standard logistic sigmoid.
  - Recommended sigmoid:
    \[ f(x) = 1.7159 \tanh \left( \frac{x}{2} \right) \]
  - When used with transformed inputs, the variance of the outputs will be close to 1.

\[ \tanh(x) = 2x(2a) - 1 \]
Usage

• Output nodes
  - Typically, a sigmoid or tanh function is used here.
  - Sigmoid for nice probabilistic interpretation (range $[0,1]$).
  - tanh for regression tasks

• Internal nodes
  - Historically, tanh was most often used.
  - tanh is better than sigmoid for internal nodes, since it is already centered.
  - Internally, tanh is often implemented as piecewise linear function (similar to hard tanh and maxout).
  - More recently: ReLU often used for classification tasks.

Effect of Sigmoid Nonlinearities

• Effects of sigmoid/tanh function
  - Linear behavior around 0
  - Saturation for large inputs

• If all parameters are too small
  - Variance of activations will drop in each layer
  - Sigmoids are approximately linear close to 0
  - Good for passing gradients through, but...
  - Gradual loss of the nonlinearity
    ⇒ No benefit of having multiple layers

• If activations become larger and larger
  - They will saturate and gradient will become zero

Another Note on Error Functions

• Squared error on sigmoid/tanh output function
  - Avoids penalizing “too correct” data points.
  - But: zero gradient for confidently incorrect classifications!
  ⇒ Do not use $L_2$ loss with sigmoid outputs (instead: cross-entropy)

Extension: ReLU

• Another improvement for learning deep models
  - Use Rectified Linear Units (ReLU)
    $g(a) = \max(0,a)$
  - Effect: gradient is propagated with a constant factor
    $\frac{\partial g(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases}$

• Disadvantages / Limitations
  - A certain fraction of units will remain “stuck at zero”.
    - If the initial weights are chosen such that the ReLU output is 0 for the entire training set, the unit will never pass through a gradient to change those weights.
  - ReLU has an offset bias, since its outputs will always be positive

Further Extensions

• Rectified linear unit (ReLU)
  $g(a) = \max(0,a)$

• Leaky ReLU
  $g(a) = \max(0.9a,a)$
  - Avoids stuck-at-zero units
  - Weaker offset bias

• ELU
  $g(a) = \begin{cases} a, & x < 0 \\ e^a - 1, & x \geq 0 \end{cases}$
  - No offset bias anymore
  - BUT: need to store activations
Recap: Optimization

**W**

Guideline (from LeCun et al., 1998 book chapter)
- The starting values of the weights can have a significant effect on the training process.
- Weights should be chosen randomly, but in a way that the sigmoid is primarily activated in its linear region.

Assuming that the training set has been normalized
- The recommended sigmoid function $f(x) = \tanh(\frac{2}{3}x)$ is used

The recommended sigmoid function is used

Nonlinearities
- Advanced techniques
  - Batch Normalization
  - Dropout

Motivation
- Keep in mind that the standard deviation is computed as
- Variance of neuron activations
- Apparently, this guideline was either little known or misunderstood for a long time
- A popular heuristic (also the standard in Torch) was to use
  $$\text{W} - U \cdot \frac{1}{\sqrt{\text{in}}} \cdot \frac{1}{\sqrt{\text{out}}}$$
- This looks almost like LeCun’s rule. However...
- When sampling weights from a uniform distribution $[a,b]$
  - Keep in mind that the standard deviation is computed as
    $$\sigma^2 = \frac{1}{12} (b-a)^2$$
  - If we do that for the above formula, we obtain
    $$\sigma^2 = \frac{1}{4n} \left( \frac{1}{n_{\text{in}}} \right)^2 = \frac{1}{4n_{\text{in}}}$$

$$\implies$$ Activations & gradients will be attenuated with each layer! (bad)

Initializing the Weights

**W**

- Motivation
  - The starting values of the weights can have a significant effect on the training process.
  - Weights should be chosen randomly, but in a way that the sigmoid is primarily activated in its linear region.

- Guideline (from LeCun et al., 1998 book chapter)
  - Assuming that the training set has been normalized
    - The recommended sigmoid function $f(x) = \tanh(\frac{2}{3}x)$ is used
    - The initial weights should be randomly drawn from a distribution (e.g., uniform or Normal) with mean zero and variance $\sigma^2 = \frac{1}{n_{\text{in}}}$
      where $n_{\text{in}}$ is the fan-in (#connections into the node).

Historical Sidenote

- Apparently, this guideline was either little known or misunderstood for a long time
  - A popular heuristic (also the standard in Torch) was to use
    $$\text{W} - U \cdot \frac{1}{\sqrt{\text{in}}} \cdot \frac{1}{\sqrt{\text{out}}}$$
  - This looks almost like LeCun’s rule. However...
  - When sampling weights from a uniform distribution $[a,b]$
    - Keep in mind that the standard deviation is computed as
      $$\sigma^2 = \frac{1}{12} (b-a)^2$$
    - If we do that for the above formula, we obtain
      $$\sigma^2 = \frac{1}{4n} \left( \frac{1}{n_{\text{in}}} \right)^2 = \frac{1}{4n_{\text{in}}}$$

$$\implies$$ Activations & gradients will be attenuated with each layer! (bad)

Analysis

- Variance of neuron activations
  - Suppose we have an input $X$ with $n$ components and a linear neuron with random weights $W$ that splits out a number $Y$.
  - What is the variance of $Y$?
    $$Y = W_1X_1 + W_2X_2 + \cdots + W_nX_n$$
  - If inputs and outputs have both mean 0, the variance is
    $$\text{Var}(W_1X_1) = \text{Var}(W_1)\text{Var}(X_1)$$
  - If we do the same for the backpropagated gradient, we get
    $$\text{Var}(W_1) = \frac{1}{n_{\text{out}}}$$
  - As a compromise, Glorot & Bengio proposed to use
    $$\text{Var}(W_1) = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

$$\implies$$ Randomly sample the weights with this variance. That’s it.
Sidenote

- When sampling weights from a uniform distribution \([a, b]\):
  - Again keep in mind that the standard deviation is computed as 
    \[ \sigma^2 = \frac{1}{12}(b - a)^2 \]
  - Glorot initialization with uniform distribution
    \[ W \sim U \left( -\frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}} \right) \]
  - Or when only taking into account the fan-in
    \[ W \sim U \left( -\frac{\sqrt{\text{fan}_{\text{in}}}}{\sqrt{\text{fan}_{\text{in}}}}, \frac{\sqrt{\text{fan}_{\text{in}}}}{\sqrt{\text{fan}_{\text{in}}}} \right) \]
  - If this had been implemented correctly in Torch from the beginning, the Deep Learning revolution might have happened a few years earlier...

Extension to ReLU

- Important for learning deep models
  - Rectified Linear Units (ReLU)
    \[ g(a) = \max(0, a) \]
  - Effect: gradient is propagated with a constant factor
    \[ \frac{\partial g(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases} \]
  - We can also improve them with proper initialization
    - However, the Glorot derivation was based on tanh units, linearity assumption around zero does not hold for ReLU.
    - He et al. made the derivations, derived to use instead

Topics of This Lecture

- Recap: Optimization
  - Effect of optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization
- Nonlinearities
- Initialization
- Advanced techniques
  - Batch Normalization
  - Dropout

Batch Normalization [Ioffe & Szegedy '14]

- Motivation
  - Optimization works best if all inputs of a layer are normalized.
- Idea
  - Introduce intermediate layer that centers the activations of the previous layer per minibatch.
  - i.e., perform transformations on all activations and undo those transformations when backpropagating gradients
  - Complication: centering + normalization also needs to be done at test time, but minibatches are no longer available at that point.
    - Learn the normalization parameters to compensate for the expected bias of the previous layer (usually a simple moving average)
- Effect
  - Much improved convergence (but parameter values are important!)
  - Widely used in practice

Dropout [Srivastava, Hinton '12]

- Idea
  - Randomly switch off units during training (a form of regularization).
  - Change network architecture for each minibatch, effectively training many different variants of the network.
  - When applying the trained network, multiply activations with the probability that the unit was set to zero during training.
    - Greatly improved performance

References and Further Reading

- More information on many practical tricks can be found in Chapter 1 of the book

Yann LeCun, Leon Bottou, Genevieve B. Orr, Klaus-Robert Mueller
References

- **ReLu**

- **Initialization**

References and Further Reading

- **Batch Normalization**

- **Dropout**