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

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Recap: Learning with Hidden Units

- How can we train multi-layer networks efficiently?
  - Need an efficient way of adapting all weights, not just the last layer.

- Idea: Gradient Descent
  - Set up an error function
    \[ E(W) = \sum_{n} L(t_n, y(x_n; W)) + \lambda \Omega(W) \]
  - E.g., \( L(t, y(x; W)) = \sum_{n} (y(x_n; W) - t_n)^2 \) \( L_2 \) loss
  \[ \Omega(W) = \|W\|^2 \]
  - \( L_2 \) regularizer ("weight decay")
  \[ \Rightarrow \text{Update each weight } W_{ij} \text{ in the direction of the gradient} \]

Recap: Backpropagation Algorithm

- Core steps
  1. Convert the discrepancy between each output and its target value into an error derivate.
  \[ \frac{\partial E}{\partial y_j} = -(t_j - y_j) \]
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.
  3. Use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights.
Recap: Backpropagation Algorithm

\[
\frac{\partial E}{\partial z_j} = \sum_j \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_j} = \sum_j \frac{\partial E}{\partial y_i} w_{ji} = \frac{\partial E}{\partial y_i} (1 - y_i) \frac{\partial E}{\partial y_i}
\]

- Efficient propagation scheme
  - \(y_i\) is already known from forward pass! (Dynamic Programming)
  - Propagate back the gradient from layer \(j\) and multiply with \(y_i\).

Recap: MLP Backpropagation Algorithm

- Forward Pass
  \[y^{(0)} = x\]
  \[\text{for } k = 1, \ldots, l \text{ do}\]
  \[z^{(k)} = W^{(k)} y^{(k-1)}\]
  \[y^{(k)} = g_k(u^{(k)})\]
  \[\text{endfor}\]
  \[y = y^{(l)}\]
  \[E = L(t, y) + \lambda \Omega(W)\]

- Backward Pass
  \[h \leftarrow \frac{\partial E}{\partial y} = \frac{\partial E}{\partial z} L(t,y) + \lambda \frac{\partial \Omega(W)}{\partial y}\]
  \[\text{for } k = l, l-1, \ldots, 1 \text{ do}\]
  \[h \leftarrow h + g'(y^{(k)})\]
  \[\frac{\partial E}{\partial W^{(k)}} = h y^{(k-1)T} + \lambda \frac{\partial \Omega(W)}{\partial W^{(k)}}\]
  \[\text{endfor}\]

- Notes
  - For efficiency, an entire batch of data \(X\) is processed at once.
  - \(\bar{\cdot}\) denotes the element-wise product.

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Computational Graphs

- We can think of mathematical expressions as graphs
  - E.g., consider the expression
  \[e = (a + b) \cdot (b + 1)\]
  - We can decompose this into the operations
  \[c = a + b\]
  \[d = b + 1\]
  \[e = c \cdot e\]
  
- General rule: sum over all possible paths from \(Y\) to \(X\) and multiply the derivatives on each edge of the path together.

Factoring Paths

- Problem: Combinatorial explosion
  - Example:
  - There are 3 paths from \(X\) to \(Y\) and 3 more from \(Y\) to \(Z\).
  - If we want to compute \(\frac{\partial Z}{\partial X}\), we need to sum over 3 \times 3 paths:
  \[
  \frac{\partial Z}{\partial X} = \alpha \delta + \alpha \epsilon + \alpha \zeta + \beta \eta + \beta \zeta + \gamma \delta + \gamma \epsilon + \gamma \zeta
  \]
  - Instead of naively summing over paths, it's better to factor them
  \[
  \frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) \cdot (\delta + \epsilon + \zeta)
  \]

Efficient Factored Algorithms

- Forward Mode Differentiation \(\frac{\partial}{\partial X}\)
  \[\Delta x = 1\]
  \[\Delta z = \Delta x \cdot \frac{\partial z}{\partial x}\]
  
- Reverse Mode Differentiation \(\frac{\partial}{\partial \theta}\)
  \[\Delta \theta = \Delta z \cdot \frac{\partial \theta}{\partial z}\]
  \[\Delta x = 1\]

- Efficient algorithms for computing the sum
  - Instead of summing over all of the paths explicitly, compute the sum more efficiently by merging paths back together at every node.
Why Do We Care?

- Let's consider the example again
  - Using forward-mode differentiation from \( b \) up...
    - Runtime: \( O(\#\text{edges}) \)
    - Result: derivative of every node with respect to \( b \).

**Slide inspired by Christopher Olah**

Image source: Christopher Olah, colah.github.io

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Obtaining the Gradients

- Approach 4: Automatic Differentiation

  Convert the network into a computational graph. Each new layer/module just needs to specify how it affects the forward and backward passes. Apply reverse-mode differentiation. Very general algorithm, used in today’s Deep Learning packages.

**Slide inspired by Christopher Olah**

Image source: Christopher Olah, colah.github.io

Modular Implementation

- Solution in many current Deep Learning libraries
  - Provide a limited form of automatic differentiation
  - Restricted to “programs” composed of “modules” with a predefined set of operations.
- Each module is defined by two main functions
  1. Computing the outputs \( y \) of the module given its inputs \( x \)
     \[
     y = \text{module}.\text{fprop}(x)
     \]
     where \( x, y \), and intermediate results are stored in the module.
  2. Computing the gradient \( \partial E/\partial x \) of a scalar cost w.r.t. the inputs \( x \) given the gradient \( \partial E/\partial y \) w.r.t. the outputs \( y \)
     \[
     \frac{\partial E}{\partial x} = \text{module}.\text{bprop}(\frac{\partial E}{\partial y})
     \]

**Slide inspired by Christopher Olah**

Image source: Christopher Olah, colah.github.io

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization
Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs
  \[ E(w) = - \sum_{n=1}^{N} \sum_{k=1}^{K} \left[ I(t_n = k) \ln \frac{\exp(w_k x_n)}{\sum_{j=1}^{J} \exp(w_j x_n)} \right] \]

- Practical issue
  - Exponentials get very big and can have vastly different magnitudes.
    - **Trick 1**: Do not compute first softmax, then log, but instead directly evaluate \( \log(\text{exp}(\text{something})) \) in the denominator.
    - **Trick 2**: Softmax has the property that for a fixed vector \( b \)
      \[ \text{softmax}(a + b) = \text{softmax}(a) \]
      \( \Rightarrow \) Subtract the largest weight vector \( w \), from the others.

---

Gradient Descent

- Two main steps
  1. Computing the gradients for each weight (last lecture)
  2. Adjusting the weights in the direction of the gradient (today)

- Recall: Basic update equation
  \[ w^{(r+1)} = w^{(r)} - \eta \frac{\partial E(w)}{\partial w_{kj}} \]

- Main questions
  - On what data do we want to apply this?
  - How should we choose the step size \( \eta \) (the learning rate)?
  - In which direction should we update the weights?

---

Stochastic vs. Batch Learning

- Batch learning
  - Process the full dataset at once to compute the gradient.
  \[ w_{kj}^{(r+1)} = w_{kj}^{(r)} - \eta \frac{\partial E(w)}{\partial w_{kj}} \]

- Stochastic learning
  - Choose a single example from the training set.
  - Compute the gradient only based on this example
  \[ w_{kj}^{(r+1)} = w_{kj}^{(r)} - \eta \frac{\partial E_{n}(w)}{\partial w_{kj}} \]

  - This estimate will generally be noisy, which has some advantages.

---

Minibatches

- Idea
  - Process only a small batch of training examples together
  - Start with a small batch size & increase it as training proceeds.

- Advantages
  - Gradients will more stable than for stochastic gradient descent, but still faster to compute than with batch learning.
  - Take advantage of redundancies in the training set.
  - Matrix operations are more efficient than vector operations.

- Caveat
  - Error function should be normalized by the minibatch size, s.t. we can keep the same learning rate between minibatches
  \[ E(W) = \frac{1}{N} \sum_{n} L(t_n, y(x_n; W)) + \frac{1}{N} \Omega(W) \]
Choosing the Right Learning Rate

• Analyzing the convergence of Gradient Descent
  » Consider a simple 1D example first
  \[ W^{t+1} = W^t - \eta \frac{dE(W)}{dW} \]
  » What is the optimal learning rate \( \eta_{opt} \)?
  
  If \( E \) is quadratic, the optimal learning rate is given by the inverse of the Hessian:
  \[ \eta_{opt} = \left( \frac{d^2E(W^t)}{dW^2} \right)^{-1} \]
  » What happens if we exceed this learning rate?

Learning Rate vs. Training Error

• Batch vs. Stochastic Learning
  » Batch Learning
    » Simplest case: steepest decent on the error surface.
    » Updates perpendicular to contour lines
  » Stochastic Learning
    » Simplest case: zig-zag around the direction of steepest descent.
    » Updates perpendicular to constraints from training examples.
Why Learning Can Be Slow

- If the inputs are correlated
  - The ellipse will be very elongated.
  - The direction of steepest descent is almost perpendicular to the direction towards the minimum!

This is just the opposite of what we want!

The Momentum Method

- Idea
  - Instead of using the gradient to change the position of the weight "particle", use it to change the velocity.

- Intuition
  - Example: Ball rolling on the error surface
  - It starts off by following the error surface, but once it has accumulated momentum, it no longer does steepest decent.

- Effect
  - Dampen oscillations in directions of high curvature by combining gradients with opposite signs.
  - Build up speed in directions with a gentle but consistent gradient.

The Momentum Method: Implementation

- Change in the update equations
  - Effect of the gradient: increment the previous velocity, subject to a decay by $\alpha < 1$.
    \[ v(t) = \alpha v(t-1) - \varepsilon \frac{\partial E}{\partial w}(t) \]
  - Set the weight change to the current velocity
    \[ \Delta w = v(t) \]
    \[ = \alpha v(t-1) - \varepsilon \frac{\partial E}{\partial w}(t) \]
    \[ = \alpha \Delta w(t-1) - \varepsilon \frac{\partial E}{\partial w}(t) \]

The Momentum Method: Behavior

- Behavior
  - If the error surface is a tilted plane, the ball reaches a terminal velocity
    \[ v(\infty) = \frac{1}{1 - \alpha} \left( -\frac{\partial E}{\partial w} \right) \]
    - If the momentum $\alpha$ is close to 1, this is much faster than simple gradient descent.
  - At the beginning of learning, there may be very large gradients.
    - Use a small momentum initially (e.g., $\alpha = 0.5$).
    - Once the large gradients have disappeared and the weights are stuck in a ravine, the momentum can be smoothly raised to its final value (e.g., $\alpha = 0.90$ or even $\alpha = 0.99$).
  - This allows us to learn at a rate that would cause divergent oscillations without the momentum.

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.

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

Behavior in a Long Valley

Behavior around a Saddle Point

Visualization of Convergence Behavior

Trick: Patience

- Saddle points dominate in high-dimensional spaces!

\[ \begin{align*}
\text{Learning often doesn’t get stuck, you just may have to wait...} \\
\end{align*} \]
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 (to be continued in the next lecture…)

Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Minibatches
  - Choosing Learning Rates
  - Momentum
  - RMS Prop
  - Other Optimizers
- Tricks of the Trade
  - Shuffling
  - Data Augmentation
  - Normalization

Shuffling the Examples

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

- 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.
Perceptual and Sensory Augmented Computing

Practical Advice

APPLY ALL
THE AUGMENTATIONS

Normalization

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

Normalization of 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).

References and Further Reading

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

G. Montavon, G. B. Orr, K.-R. Müller (Eds.)
Neural Networks: Tricks of the Trade

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