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
  - 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
    \[ E(W) = \sum_{i} L(t_i, y(x_i; W)) + \lambda \Omega(W) \]
  - with a loss \( L(\cdot) \) and a regularizer \( \Omega(\cdot) \).
  - E.g., \( L(t_i, y(x_i; W)) = \sum_{i} (y(x_i; W) - t_i)^2 \)
  - \( \Omega(W) = \|W\|_2^2 \)
  - \( L_2 \) regularizer
  - ("weight decay")

- Update each weight \( W_{ij}^{(l)} \) in the direction of the gradient \( \Delta W_{ij}^{(l)} \).

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation
  - Compute the gradients for each variable analytically.
  - What is the problem when doing this?
Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions
  \[
  \Delta z = \frac{dz}{dy} \Delta y = \frac{dz}{dx} \Delta x = \frac{dz}{dy} \frac{dy}{dx} \Delta x
  \]

- Multi-dimensional case: Total derivative
  \[
  \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x} + \ldots = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
  \]

  \[\Rightarrow\] Need to sum over all paths that lead to the target variable \(x\).

---

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation
  - Compute the gradients for each variable analytically.
  - What is the problem when doing this?
    \[\Rightarrow\] With increasing depth, there will be exponentially many paths!
    \[\Rightarrow\] Infeasible to compute this way.

- Approach 2: Numerical Differentiation
  - Given the current state \(W^{(k)}\), we can evaluate \(E(W^{(k)})\).
  - Idea: Make small changes to \(W^{(k)}\) and accept those that improve \(E(W^{(k)})\).
  - Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

- Approach 3: Incremental Analytical Differentiation
  - Idea: Compute the gradients layer by layer.
    - Each layer below builds upon the results of the layer above.
    \[\Rightarrow\] The gradient is propagated backwards through the layers.
    \[\Rightarrow\] Backpropagation algorithm

---

Backpropagation Algorithm

- Core steps
  1. Convert the discrepancy between each output and its target value into an error derivate.
  \[E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2\]
  \[\frac{\partial E}{\partial y_j} = -(t_j - y_j)\]
  
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.
  \[\frac{\partial E}{\partial y_j} \rightarrow \frac{\partial E}{\partial w_{ji}}\]

  3. Use error derivatives w.r.t. activities to get error derivatives w.r.t. the incoming weights.
Backpropagation Algorithm

\[ \frac{\partial E}{\partial y_j} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial y_j} \]

Connections:
\[ z_j^{(k)} = \sum w_{ij}^{(k)} y_i^{(k-1)} \]
\[ y_j^{(k)} = g(z_j^{(k)}) \]

\[ \frac{\partial E}{\partial w_{ij}^{(k)}} = \frac{\partial E}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} \]

\[ \frac{\partial E}{\partial b_j^{(k)}} = \frac{\partial E}{\partial z_j^{(k)}} \frac{\partial z_j}{\partial b_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial b_j} \]

Efficient propagation scheme
\[ y_{(k-1)} = \{ y_j^{(k-1)} \} \]
\[ \frac{\partial E}{\partial y_j^{(k-1)}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} \frac{\partial z_j}{\partial y_j^{(k-1)}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial y_j^{(k-1)}} \]

Summary: MLP Backpropagation

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

- **Backward Pass**
  \[ h \leftarrow \frac{\partial E}{\partial y} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial h} + \lambda \frac{\partial \Omega}{\partial W} \]
  for \( k = l, l-1, \ldots, 1 \) do
  \[ h \leftarrow \frac{\partial E}{\partial W^{(k)}} = h \odot g'(z^{(k)}) \frac{\partial E}{\partial z^{(k)}} + \lambda \frac{\partial \Omega}{\partial W^{(k)}} \]
  \[ h \leftarrow \frac{\partial E}{\partial b^{(k)}} = W^{(k) \top} h \]
  endfor

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

Analysis: Backpropagation

- **Backpropagation is the key to make deep NNs tractable**
  - However...
  - **The Backprop algorithm given here is specific to MLPs**
    - It does not work with more complex architectures, e.g. skip connections or recurrent networks!
    - Whenever a new connection function induces a different functional form of the chain rule, you have to derive a new Backprop algorithm for it. \( \Rightarrow \) Tedious...
  - **Let’s analyze Backprop in more detail**
    - This will lead us to a more flexible algorithm formulation
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
    \[ c = (a + b) \times (b + 1) \]
  - We can decompose this into the operations
    \[ a + b \quad d = b + 1 \quad e = c \times d \]
  - and visualize this as a computational graph.
- Evaluating partial derivatives \( \frac{\partial c}{\partial X} \) in such a graph
  - 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:
    \[ X \xrightarrow{a} Y \xrightarrow{\beta} Z \]
    \[ \frac{\partial Z}{\partial X} = \alpha \delta + \alpha \zeta + \beta \delta + \beta \zeta + \gamma \delta + \gamma \zeta \]
    Instead of naively summing over paths, it’s better to factor them
    \[ \frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) \times (\delta + \epsilon + \zeta) \]

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 \( b \).
  
  \[ \frac{\partial c}{\partial b} = b + 1 \]

Efficient Factored Algorithms

- 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 reverse-mode differentiation from \( c \) down...
    - Runtime: \( O(\#\text{edges}) \)
    - Result: derivative of \( c \) with respect to every node.
  
  \[ \frac{\partial c}{\partial b} = b + 1 \]
Topics of This Lecture

- Learning Multi-layer Networks
  - Recap: Backpropagation
  - Computational graphs
  - Automatic differentiation
  - Practical issues
- Gradient Descent
  - Stochastic Gradient Descent & Mini-batches
  - 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

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 \( x \)
     \[ y = \text{module}.\text{fp}rop(x) \]
     where \( x \), \( y \), and intermediate results are stored in the module.
  2. Computing the gradient \( \frac{\partial E}{\partial x} \) of a scalar cost w.r.t. the inputs \( x \) given the gradient \( \frac{\partial E}{\partial y} \) w.r.t. the outputs \( y \)
     \[ \frac{\partial E}{\partial x} = \text{module}.\text{bp}rop \left( \frac{\partial E}{\partial y} \right) \]

Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs
  \[ E(w) = - \sum_{n=1}^{N} \sum_{k=1}^{K} \left( I(t_n = k) \ln \frac{\exp(w_k^T x)}{\sum_{j=1}^{K} \exp(w_j^T x)} \right) \]
- 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-exp in the nominator and log-sum-exp 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 \( w_j \) from the others.
Gradient Descent

• Two main steps
  1. Computing the gradients for each weight
  2. Adjusting the weights in the direction of the gradient

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

• 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.

• Stochastic learning
  ➢ Choose a single example from the training set.
  ➢ Compute the gradient only based on this example
  ➢ This estimate will generally be noisy, which has some advantages.

• Batch learning advantages
  ➢ Conditions of convergence are well understood.
  ➢ Many acceleration techniques (e.g., conjugate gradients) only operate in batch learning.
  ➢ Theoretical analysis of the weight dynamics and convergence rates are simpler.

• Stochastic learning advantages
  ➢ Usually much faster than batch learning.
  ➢ Often results in better solutions.
  ➢ Can be used for tracking changes.

• Middle ground: Minibatches

Choosing the Right Learning Rate

• Analyzing the convergence of Gradient Descent
  ➢ Consider a simple 1D example first
  \[ W^{(r+1)} = W^{(r)} - \frac{\partial E(W)}{\partial W} \]
  ➢ What is the optimal learning rate \( \eta_{opt} \)?

  ➢ If \( E \) is quadratic, the optimal learning rate is given by the inverse of the Hessian
  \[ \eta_{opt} = \left( \frac{\partial^2 E(W)}{\partial W^2} \right)^{-1} \]
  ➢ What happens if we exceed this learning rate?
Choosing the Right Learning Rate

- Behavior for different learning rates

Learning Rate vs. Training Error

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

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!

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 descent.
- 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 $\alpha < 1$.
    \[
    v(t) = \alpha v(t-1) - \frac{\partial E}{\partial w}(t)
    \]
  - Set the weight change to the current velocity
    \[
    \Delta w = v(t) = \alpha v(t-1) - \frac{\partial E}{\partial w}(t)
    \]

- 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.

- 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 $\sqrt{\text{MeanSq}(w_{ij}, t)}$.

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

Reducing the Learning Rate

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…)

- 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.

- Saddle points dominate in high-dimensional spaces!

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

- Reduced learning rate

- Training error

- Epoch

- Reduced learning rate

- Norm of the gradients
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.
  - 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

- 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.

Practical Advice

- Motivation
  - Consider the Gradient Descent update steps
    \[ w_{kj}^{(r+1)} = w_{kj}^{(r)} - \eta \frac{\partial E(w)}{\partial w_{kj}} \]
    - From backpropagation, we know that
      \[ \frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{kj}} \]
    - 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
  - When working with stochastic gradient descent or minibatches, make use of shuffling.
Normalizing 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

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