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: Automatic Differentiation

- Approach for obtaining the gradients

  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

Correction: Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs

  $$E(w) = - \sum_{n=1}^{N} \sum_{k=1}^{K} \left( t_n = k \right) \ln \left( \frac{\exp(w_k^T x)}{\sum_{j=1}^{K} \exp(w_j^T x)} \right)$$

- 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-sum-exp in the nominator and log-sum-exp in the denominator.
    - Trick 2: Softmax has the property that for a fixed vector $b$
      $\text{softmax}(a + b) = \text{softmax}(a)$

      Subtract the largest weight vector $w$ from the others.
Recap: Choosing the Right Learning Rate

- Convergence of Gradient Descent
  - Simple 1D example:
    \[ W^{(r+1)} = W^{(r)} - \eta \frac{dE[W]}{dW} \]
  - What is the optimal learning rate \( \eta_{opt} \)?
  - If \( E \) is quadratic, the optimal learning rate is given by the inverse of the Hessian:
    \[ \eta_{opt} = \left( \frac{d^2E[W(t)]}{dt^2} \right)^{-1} \]
  - Advanced optimization techniques try to approximate the Hessian by a simplified form.
  - If we exceed the optimal learning rate, bad things happen!

Recap: Advanced Optimization Techniques

- Momentum
  - Instead of using the gradient to change the position of the weight "particle", use it to change the velocity.
  - Effect: dampen oscillations in directions of high curvature
  - Nesterov-Momentum: Small variation in the implementation
- RMS-Prop
  - Separate learning rate for each weight: Divide the gradient by a running average of its recent magnitude.
  - Some more recent techniques, work better for some problems. Try them.
  - AdaGrad
  - AdaDelta
  - Adam

Example: Behavior in a Long Valley

Example: Behavior around a Saddle Point

Visualization of Convergence Behavior

Trick: Patience

- Saddle points dominate in high-dimensional spaces!
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

- Augmented training data (from one original image)
Practical Advice

**APPLY ALL**

**THE AUGMENTATIONS**

Normalization

- **Motivation**
  - Consider the Gradient Descent update steps
  
  \[
  w_{kj}^{t+1} = w_{kj}^t - \eta \frac{\partial E(w)}{\partial w_{kj}}
  \]

  - From backpropagation, we know that
  
  \[
  \frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}}
  \]

  - 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.
  
  \Rightarrow \text{Weights can only all increase or decrease together.}

  \Rightarrow \text{Slow convergence}

Normalization

- **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-Loeve 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:
  
  \[
  f(x) = 1.7159 \tanh \left( \frac{2}{3}x \right)
  \]

  \Rightarrow \text{When used with transformed inputs, the variance of the outputs will be close to 1.}

Topics of This Lecture

- **Recap: Optimization**
  - Effect of optimizers

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

- **Nonlinearities**
  - Initialization

- **Advanced techniques**
  - Batch Normalization
  - Dropout

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)
  - Effect: gradient is propagated with a constant factor
    \[ \frac{\partial y(a)}{\partial a} = \begin{cases} 1, & a > 0 \\ 0, & \text{else} \end{cases} \]

- Advantages
  - Much easier to propagate gradients through deep networks.
  - We do not need to store the ReLU output separately
    - Reduction of the required memory by half compared to tanh!

  ⇒ ReLU has become the de-facto standard for deep networks.

Further Extensions

- Rectified linear unit (ReLU)
  \[ g(a) = \max(0,a) \]

- Leaky ReLU
  \[ g(a) = \max(fix,a) \]
  - 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

Topics of This Lecture

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

**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 \( f(x) = \frac{1}{1 + e^{-x}} \) 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
  \[
  W \sim \mathcal{U}(0, \frac{1}{ \sqrt{\text{fan-in}}})
  \]
  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
    \[
    \sigma^2 = \frac{1}{n_{\text{in}}} \Rightarrow \text{Activations & gradients will be attenuated with each layer! (bad)}
    \]

Glorot Initialization

**Breakthrough results**
- In 2010, Xavier Glorot published an analysis of what went wrong in the initialization and derived a more general method for automatic initialization.
- This new initialization massively improved results and made direct learning of deep networks possible overnight.

Analysis

**Variance of neuron activations**
- Suppose we have an input \( X \) with \( n \) components and a linear neuron with random weights \( W \) that spits out a number \( Y \).
- What is the variance of \( Y \)?
- If inputs and outputs have both mean 0, the variance is
- \[
  \text{Var}(W_i X_i) = E[X_i^2]\text{Var}(W_i) + E[W_i^2]\text{Var}(X_i) + \text{Var}(W_i)\text{Var}(i) = \text{Var}(W_i)\text{Var}(X_i)
  \]
- If the \( X_i \) and \( W_i \) are all i.i.d, then
- \[
  \text{Var}(Y) = \text{Var}(W_i X_i + W_j X_j + \cdots + W_n X_n) = n\text{Var}(W_i)\text{Var}(X_i)
  \]
- The variance of the output is the variance of the input, but scaled by \( n \) \( \text{Var}(W_i) \).

Analysis (cont’d)

**Variance of neuron activations**
- If we want the variance of the input and output of a unit to be the same, then \( n \) \( \text{Var}(W_i) \) should be 1. This means
- \[
  \text{Var}(W_i) = \frac{1}{n_{\text{out}}}
  \]
- If we do the same for the backpropagated gradient, we get
- \[
  \text{Var}(W_i) = \frac{1}{n_{\text{in}}}
  \]
- As a compromise, Glorot & Bengio proposed to use
- \[
  \text{Var}(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}}
  \]
- 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
  \[
  W \sim \mathcal{U}\left(0, \frac{\sqrt{5}}{\sqrt{n_{\text{in}}} + n_{\text{out}}}, \frac{\sqrt{5}}{\sqrt{n_{\text{in}}} + n_{\text{out}}}ight)
  \]
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.
  - Change network architecture for each data point, 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.
  - Greatly improved performance

References and Further Reading

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


References

- ReLU
- Initialization
References and Further Reading

- **Batch Normalization**

- **Dropout**