SDU Summer School

Deep Learning

Summer 2022

Welcome to the Summer School

Optimization for Deep Learning

- Parameter Initialization
- Batch Normalization
- Pre-Training

Types of Initialization

1. Non-iterative optimization requires no initialization
   - Simply solve for solution point

2. Iterative but converge regardless of initialization
   - Acceptable solutions in acceptable time

3. Iterative but affected by choice of Initialization
   - Deep learning training algorithms are iterative
   - Initialization determines whether it converges at all
   - Can hugely determine how quickly learning converges

Modern Initialization Strategies

- They are simple and heuristic
- Based on achieving nice properties
- But problem is a difficult one
  - Some initial points are beneficial for optimization but detrimental to generalization

- Only property known with certainty: Initial parameters must be chosen to **break symmetry**
  - If two hidden units have the same inputs and same activation function, then they must have different initial parameters
  - Usually best to initialize each unit to compute a different function
  - This motivates use random initialization of parameters

Choice of biases

- Biases for each unit are heuristically chosen constants
- Only the weights are initialized randomly
- Extra parameters such as conditional variance of a prediction are constants like biases

Weights drawn from Gaussian

- Weights are almost always drawn from a Gaussian or uniform distribution
  - Choice of Gaussian or uniform does not seem to matter much but not studied exhaustively

- Scale of the initial distribution does have an effect on outcome of optimization and ability to generalize
  - Larger initial weights will yield stronger symmetry-breaking effect, helping avoid redundant units
  - Too large may result in exploding values

Heuristics for initial scale of weights

- One heuristic is to initialize the weights of a fully connected layer with N inputs and M outputs by sampling each weights from $\mathrm{Uniform}(-r, r)$ with

$$r = \frac{1}{\sqrt{N}}$$

- Another heuristic is normalized initiation with

$$r = \sqrt{\frac{6}{N + M}}$$

  - Which is a compromise between the goal of initializing all layers to have the same activation variance and the goal of having all layers having the same gradient variance

Initialization for the biases

- Bias settings must be coordinated with setting weights

- Setting biases to zero is compatible with most weight initialization schemes

- Situations for nonzero biases:
  - Bias for an output unit: initialize to obtain right marginal statistics for output
    - Set bias to inverse of activation function applied to the marginal statistics of the output in the training set
    - Assuming that the weights in the beginning are so small, that output is driven only by biases
  - Choose bias to causing too much saturation at initialization

Optimization for Deep Learning

- Parameter Initialization
- Batch Normalization
- Pre-Training

Batch Normalization

- Batch normalization: exciting recent innovation
- Method is to replace activations with zero-mean with unit variance activations

- Motivation is difficulty of choosing learning rate ϵ in deep networks
  - Method adds an additional step between layers, in which the output of the earlier layer is normalized
    - By standardizing the mean and standard deviation of each individual unit
  - It is a method of adaptive re-parameterization
    - It is not an optimization algorithm at all
    - A method to reduce internal covariate shift in neural networks

Motivation: Difficulty of composition

- Very deep models involve compositions of several functions or layers

$$f(\boldsymbol{x}, \boldsymbol{w}) = f^{(l)}\left(\dots f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)\right)$$

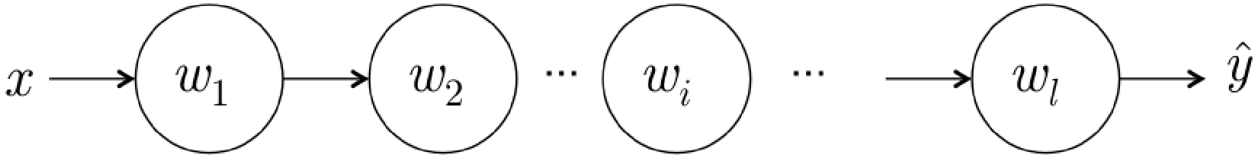
- The gradient tells us how to update each parameter

$$\boldsymbol{w}^{\tau+1} = \boldsymbol{w}^{\tau} - \epsilon \nabla_{\boldsymbol{w}} J(f(\boldsymbol{x}, \boldsymbol{w}), y)$$

  - **Under assumption that other layers do not change**
  - **BUT**: We update all l layers simultaneously
  - When we make the change unexpected results can happen
  - Because many functions are changed simultaneously, effects can accumulate

Choosing learning rate ϵ in multilayer

- Simple example:
  - l layers, one multiplication unit per layer, no activation function



  - Network simply computes

$$\hat{y} = x \cdot w_1 \cdot w_2 \ldots \cdot w_l$$

  - Output of Layer i is $h_i = h_{i-1} w_i$
  - Output is a linear function of input x but a nonlinear function of the weights w_i

Gradient in Simple example

- Suppose our cost function has put a gradient of 1 on $\hat{y}$, so we wish to decrease $\hat{y}$ slightly.

- The back-propagation algorithm can then compute a gradient
$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} \hat{y}$$
  - Which corresponds to the gradient $\boldsymbol{g}$ evaluated at $\mathrm{y} = \hat{y}$

- When using the update $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g}$ predicts that $\hat{y}$ decreases by $\epsilon \boldsymbol{g}^T \boldsymbol{g}$
- If we want to decrease $\hat{y}$ by 0.1, we could set the learning rate to
$$\epsilon = \frac{0.1}{\boldsymbol{g}^T \boldsymbol{g}}$$

Difficulty of Multilayer learning Rate

- With the first order information, we would set the learning rate to

$$\epsilon = \frac{0.1}{\boldsymbol{g}^T \boldsymbol{g}}$$

- Problem: We have to deal with many 2nd, 3rd … effects. The new value in fact is

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l)$$

  - One example for a second order term would be $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$
  - This term might be negligible if $\prod_{i=3}^{l} w_i$ is small or exponentially large if weights are larger than 1.

- This makes it very hard to choose ε because the effects of an update for one layer depend so strongly on all other layers
  - Higher order methods tackle this problem, but still only for small l

The Batch Normalization Solution

- Provides an elegant way of re-parameterizing almost any network
- Significantly reduces the problem of coordinating updates across many layers
- Can be applied to any input or hidden layer in a network

Batch Normalization Equations

- $\boldsymbol{H}$: minibatch of activations of layer to normalize
  - arranged as a design matrix
  - With activations for each example appearing in a row

$$H = \begin{array}{cc} \text{units} \longrightarrow & \\ \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ . & . & . \\ a_{N,1} & a_{N,2} & a_{N,3} \end{bmatrix} & \text{samples} \end{array}$$

- To normalize $\boldsymbol{H}$ we replace it with $\boldsymbol{H}' = \frac{\boldsymbol{H}-\mu}{\sigma}$
  - Where μ is a vector containing the mean of each unit and σ is the std. deviation of each unit
  - The arithmetic here is based on broadcasting the vector μ and the vector σ to be applied to every row of $\boldsymbol{H}$
  - Within each row, the arithmetic is element-wise
  - $H_{i,j}$ is normalized by subtracting μ_j & dividing by σ_j

Normalization Details

- Rest of the network operates on ***H'*** in the same way that the original network operated on ***H***
- At training time

$$\mu = \frac{1}{m}\sum_i H_i \;\; and \;\; \sigma = \sqrt{\delta + \frac{1}{m}\sum_i (H-\mu)_i^2}$$

- where δ is a small positive value such as 10^{-8} imposed to avoid encountering the undefined gradient of $z = 0$
  - Crucially we back propagate through these operations for computing the mean and std dev
  - And for applying them to normalize ***H***
    - This means that the gradient will never propose an operation that acts simply to increase std dev or mean of h_i the normalization operations remove the effect of such an action and aero out the component in the gradient

Batch Normalization at Test time

- At test time, μ and σ may be replaced by running averages that were collected during training time

- This allows the model to be evaluated on a single example without needing to use definitions of μ and σ that depend on an entire minibatch

Revisiting the simple example

- Revisiting the $\hat{y} = x \cdot w_1 \cdot w_2 \ldots \cdot w_l$ example we can mostly resolve the difficulties in learning the model by normalizing h_{l-1}

- Suppose that x is drawn from a unit Gaussian
- Then h_{l-1} will also come from a Gaussian, because the transformation from x to h_l is linear
- However h_{l-1} will no longer have zero mean, unit variance

Restoring zero-mean unit variance

- After applying batch normalization, we obtain the normalized $\hat{h}_{l-1}$ that restores zero mean and unit variance
  - For almost any update to the lower layers, $\hat{h}_{l-1}$ will remain a unit Gaussian
  - Output ŷ may be learned as a simple linear function $\hat{y} = \boldsymbol{w}_l \hat{h}_{l-1}$

- Learning in this model is now very simple
  - Because parameters at the lower layers do not have an effect in most cases

- Their output is always renormalized to a unit Gaussian

Batch Normalization -> learning easy

- Without normalization, updates would have an extreme effect of the statistics of h_{l-1}

- Batch normalization has thus made this model easier to learn

- In this example the ease of learning came from making the lower layers useless
  - In our linear example: Lower layers not harmful but not beneficial either
    - Because we have normalized-out all effects of higher order than 1st and 2nd order stats

- In a deep neural network lower levels still can perform useful nonlinear transformations

Reintroducing Expressive Power

- Normalizing the mean and standard deviation can reduce the expressive power of the neural network containing that unit
- To maintain the expressive power replace the batch of hidden unit activations $\boldsymbol{H}'$ with $\boldsymbol{\gamma H}' + \boldsymbol{\beta}$
  - γ and β are learned parameters that allow the new variable to have any mean and standard deviation
- Why did we normalize to zero and std. dev. when we then allow all means and deviations again?
  - Has different learning dynamics than unnormalized approach
  - It has the same power as $\boldsymbol{H}$, but the values of $\boldsymbol{H}$ where determined by a complicated interaction between the parameters in the layers below
  - The new parametrization, the mean is only determined by $\boldsymbol{\beta}$

Optimization for Deep Learning

- Parameter Initialization
- Batch Normalization
- Pre-Training

Motivation

- Sometimes, directly training a model to solve a specific task can be too ambitious, if:
  - Model is too complex and hard to optimize
  - The task is very difficult

- It may be more effective to
  - Train a simpler model to solve the task, then move on to confront the final task
  - Methods collectively known as pretraining

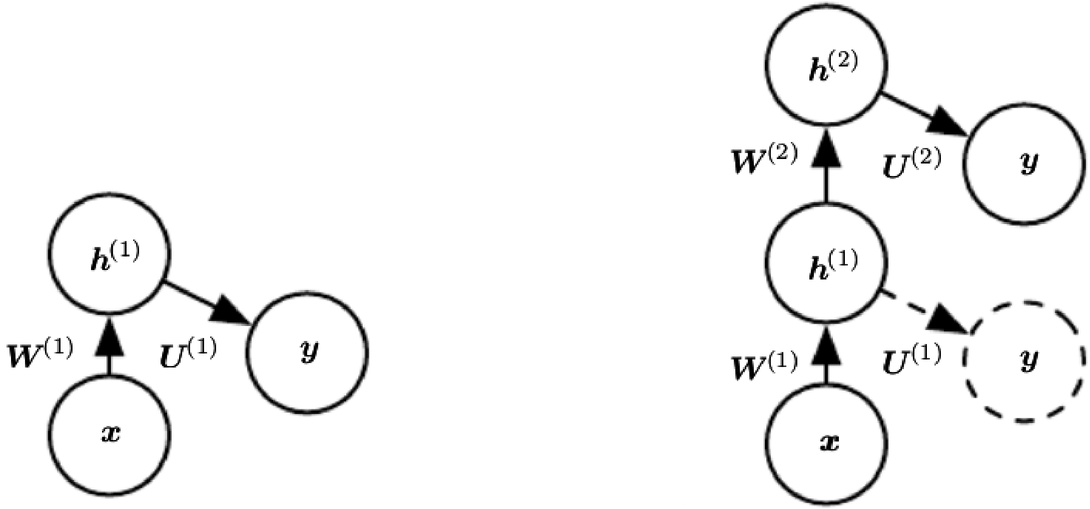
Greedy Supervised Pretraining

- Greedy Algorithm:
  1. Break a problem into many components
  2. Solve for the optimal version of each component in isolation
  3. Combine the solutions

- Combining the component solutions may not yield an optimal complete solution
- However, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution
- Quality of a greedy solution is often acceptable if not optimal
- Initializing the joint optimization algorithm with a greedy solution can speed it up and improve the quality of the solution

Training each layer separately

- Supervised learning involving only a subset of the layers in the final neural network
- An example of greedy supervised pretraining
  - In which each added hidden layer is pretrained as part of a shallow supervised MLP
  - Taking as input the output of the previously trained hidden layer



Extension to Transfer Learning

- Pretraining extends the idea to transfer learning

- Pretrain convolutional net with k layers on tasks
  - E.g. on a subset of 1000 ImageNet object categories

- Then initialize same-size network with the first k layers of the first net
  - All layers of second network (with upper layers initialized randomly) are then jointly trained to perform a different set of tasks
    - E.g. another subset of 1000 ImageNet categories, with fewer training examples than for the first set of tasks

FitNets

- While depth improves performance, it also makes gradient-based training more difficult since deeper networks are more non-linear.

- Solution is to train a network with low enough depth (e.g. 5) and great enough width (no. of units per layer) to be easy to train

- This network becomes a teacher for a second network, designated the student

- Student network is much deeper and thinner and would be difficult to train with SGD, (e.g., 15-20 layers)

Training the student network

- Task is made easier by training student network not only to predict output for original task, but also to predict value of middle layer of the teacher network

- This extra task provides a set of hints about how the hidden layers should be used and can simplify the optimization problem

- Additional parameters are introduced to regress the middle layer of the 5-layer teacher network from the middle layer of the deeper student network

Predicting Intermediate Layers

- Instead of predicting the final classification target, the objective is to predict the middle hidden layer of the teacher network

- Objectives of Lower layers of student network:
  1. Help outputs of student network accomplish task
  2. Predict intermediate layer of the teacher network

- A thin-deep network may be more difficult to train than a wide-shallow network, but may generalize better and has lower computational cost if it is thin enough to have far fewer parameters.