Introductory Overview Lecture
The Deep Learning Revolution
Part II: Optimization, Regularization

Russ Salakhutdinov

Machine Learning Department
Carnegie Mellon University
Canadian Institute for Advanced Research
Used Resources

- Some material and slides for this lecture were borrowed from
  
  - Hugo Larochelle’s class on Neural Networks:
    https://sites.google.com/site/deeplearningsummerschool2016/
  
  - Grover and Ermon IJCA-ECA Tutorial on Deep Generative Models
    https://ermongroup.github.io/generative-models/
Supervised Learning

- Given a set of labeled training examples: \( \{ \mathbf{x}^{(t)}, y^{(t)} \} \), we perform **Empirical Risk Minimization**

\[
\arg\min_{\theta} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) + \lambda \Omega(\theta)
\]

where

- \( f(\mathbf{x}^{(t)}; \theta) \) is a (non-linear) function mapping inputs to outputs, parameterized by \( \theta \) - Non-convex optimization

- \( l(f(\mathbf{x}^{(t)}; \theta), y^{(t)}) \) is the loss function

- \( \Omega(\theta) \) is a regularization term
Supervised Learning

- Given a set of labeled training examples: \( \{x^{(t)}, y^{(t)}\} \), we perform Empirical Risk Minimization

\[
\arg\min_{\theta} \frac{1}{T} \sum_{t} l(f(x^{(t)}; \theta), y^{(t)}) + \lambda \Omega(\theta)
\]

- Loss Functions:
  - For classification tasks, we can use Cross-Entropy Loss
  - For regression tasks, we can use Squared Loss
Training

- **Empirical Risk Minimization**

\[
\arg \min_{\theta} \frac{1}{T} \sum_{t} l(f(x^{(t)}; \theta), y^{(t)}) + \lambda \Omega(\theta)
\]

- **To train a neural network, we need:**
  - Loss Function: \( l(f(x^{(t)}; \theta), y^{(t)}) \)
  - A procedure to compute its gradients: \( \nabla_{\theta} l(f(x^{(t)}; \theta), y^{(t)}) \)
  - Regularizer and its gradient: \( \Omega(\theta), \nabla_{\theta} \Omega(\theta) \)
Stochastic Gradient Descent (SGD)

- Perform updates after seeing each example:
  - Initialize: \( \theta \equiv \{ W^{(1)}, b^{(1)}, \ldots, W^{(L+1)}, b^{(L+1)} \} \)
  - For \( t=1:T \)
    - for each training example \( (x^{(t)}, y^{(t)}) \)
      \[
      \Delta = -\nabla_\theta l(f(x^{(t)}; \theta), y^{(t)}) - \lambda \nabla_\theta \Omega(\theta) \\
      \theta \leftarrow \theta + \alpha \Delta
      \]

Learning rate: Difficult to set in practice
Mini-batch, Momentum

- Make updates based on a mini-batch of examples (instead of a single example):
  - The gradient is the average regularized loss for that mini-batch
  - More accurate estimate of the gradient
  - Leverage matrix/matrix operations, which are more efficient

- Momentum: Use an exponential average of previous gradients:

  \[ \overline{\nabla}_{\theta}^{(t)} = \nabla_{\theta} l(f(x^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\theta}^{(t-1)} \]

- Can get pass plateaus more quickly, by “gaining momentum”
Adapting Learning Rates

- Updates with adaptive learning rates ("one learning rate per parameter")
  - **Adagrad**: learning rates are scaled by the square root of the cumulative sum of squared gradients
    \[
    \nabla_{\theta}^{(t)} = \frac{\nabla_{\theta} l(f(x^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}
    \]
    \[
    \gamma^{(t)} = \gamma^{(t-1)} + \left( \nabla_{\theta} l(f(x^{(t)}), y^{(t)}) \right)^2
    \]
  - **RMSProp**: instead of cumulative sum, use exponential moving average
    \[
    \nabla_{\theta}^{(t)} = \frac{\nabla_{\theta} l(f(x^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}
    \]
    \[
    \gamma^{(t)} = \beta \gamma^{(t-1)} + (1 - \beta) \left( \nabla_{\theta} l(f(x^{(t)}), y^{(t)}) \right)^2
    \]
  - **Adam**: essentially combines RMSProp with momentum

(Douchi et. al, 2011, Kingma and Ba, 2014)
Regularization

\[
\arg\min_{\theta} \frac{1}{T} \sum_{t} l(f(x^{(t)}; \theta), y^{(t)}) + \lambda \Omega(\theta)
\]

- **L2 regularization:**
  \[
  \Omega(\theta) = \sum_{k} \sum_{i} \sum_{j} \left(W_{i,j}^{(k)}\right)^2 = \sum_{k} \|W^{(k)}\|_F^2
  \]

- **L1 regularization:**
  \[
  \Omega(\theta) = \sum_{k} \sum_{i} \sum_{j} |W_{i,j}^{(k)}|
  \]
Dropout

- **Key idea**: Cripple neural network by removing hidden units stochastically
  - Each hidden unit is set to 0 with probability 0.5
  - Hidden units cannot co-adapt to other units
  - Hidden units must be more generally useful
  - Could use a different dropout probability, but 0.5 usually works well

Srivastava et al., JMLR 2014
Dropout

- Use random binary masks $m^{(k)}$
  - Layer pre-activation for $k > 0$
    \[ a^{(k)}(x) = b^{(k)} + W^{(k)} h^{(k-1)}(x) \]
  - Hidden layer activation ($k = 1$ to $L$):
    \[ h^{(k)}(x) = g(a^{(k)}(x)) \odot m^{(k)} \]
  - Output activation ($k = L+1$)
    \[ h^{(L+1)}(x) = o(a^{(L+1)}(x)) = f(x) \]
Dropout at Test Time

- At test time, we replace the masks by their expectation
  - This is simply the constant vector 0.5 if dropout probability is 0.5

- Beats regular backpropagation on many datasets and has become a standard practice

- Ensemble: Can be viewed as a geometric average of exponential number of networks.
Batch Normalization

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - Could normalization be useful at the level of the hidden layers?

- **Batch normalization** is an attempt to do that (Ioffe and Szegedy, 2015)
  - each hidden unit’s pre-activation is normalized (mean subtraction, stddev division)
  - during training, mean and stddev is computed for each mini-batch
  - backpropagation takes into account the normalization
  - at test time, the global mean and stddev is used

- Why normalize the pre-activation?
  - helps keep the pre-activation in a non-saturating regime
    → helps with vanishing gradient problem

\[
a^{(k)}(x) = b^{(k)} + W^{(k)} h^{(k-1)}(x)
\]
Batch Normalization

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...x_m\}$;  
Parameters to be learned: $\gamma, \beta$

**Output:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

\[
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean} \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad // \text{mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{scale and shift}
\end{align*}
\]

Learned linear transformation to adapt to non-linear activation function ($\gamma$ and $\beta$ are trained)
Model Selection

- **Training Protocol:**
  - Train your model on the **Training Set** \( \mathcal{D}_{\text{train}} \)
  - For model selection, use **Validation Set** \( \mathcal{D}_{\text{valid}} \)
    - *Hyper-parameter search: hidden layer size, learning rate, number of iterations, etc.*
  - Estimate generalization performance using the **Test Set** \( \mathcal{D}_{\text{test}} \)

- Generalization is the behavior of the model on **unseen examples**.
Early Stopping

- To select the number of epochs, stop training when validation set error increases → Large Model can Overfit
But in Practice

- To select the number of epochs, stop training when validation set error increases → Large Model can Overfit

Implicit Regularization

- Optimization plays a crucial role in generalization
- Generalization ability is not controlled by network size but rather by some other implicit control

Behnam Neyshabur, PhD thesis 2017
Neyshabur et al., Survey Paper, 2017
Best Practice

- Given a dataset D, pick a model so that:
  - You can achieve 0 training error → Overfit on the training set.

- Regularize the model (e.g. using Dropout).

- Initialize parameters so that each feature across layers has similar variance. Avoid units in saturation.

- SGD with momentum, batch-normalization, and dropout usually works very well.
Choosing Architecture

- How can we select the right architecture:
  - Manual tuning of features is now replaced with the manual tuning of architectures

- Many hyper-parameters:
  - Number of layers, number of feature maps

- Cross Validation
- Grid Search (need lots of GPUs)
- Smarter Strategies
  - Bayesian Optimization
AlexNet

- 8 layers total
- Trained on Imagenet dataset [Deng et al. CVPR’09]
- 18.2% top-5 error

[From Rob Fergus’ CIFAR 2016 tutorial] Krizhevsky et al., NIPS 2012
AlexNet

- Remove top fully connected layer 7
- Drop ~16 million parameters
- Only 1.1% drop in performance!

[From Rob Fergus’ CIFAR 2016 tutorial]  Krizhevsky et al., NIPS 2012
AlexNet

- Remove layers 3 4,6 and 7
- Drop ~50 million parameters
- 33.5% drop in performance!

- Depth of the network is the key

[From Rob Fergus’ CIFAR 2016 tutorial] Krizhevsky et al., NIPS 2012
GoogleNet

- 24 layer model

(Szegedy et al., Going Deep with Convolutions, 2014)
Residual Networks

- Really, really deep convnets do not train well, e.g. CIFAR10:

- Key idea: introduce “pass through” into each layer

- Thus only residual now needs to be learned:

(He, Zhang, Ren, Sun, CVPR 2016)