Deep Learning Theory and Practice

Lecture 8
Practical aspects of training deep neural networks

Dr. Ted Willke
willke@pdx.edu

Wednesday, April 24, 2019
Review of Lecture 7

• Completed our discussion of growth functions

Counts the most dichotomies on any \( N \) points.

\[
m_H(N) = \max_{x_1, \ldots, x_N \in X} |H(x_1, \ldots, x_N)|
\]

Introduced break points: If no dataset of size \( k \) can be shattered by \( H \), then \( k \) is a break point.

No break point \( \implies m_H(N) = 2^N \)

Any break point \( \implies m_H(N) \) is polynomial in \( N \)

\( m_H(N) \) grows faster for more complex hypothesis sets and the break point increases.

\( H \) is positive rays \( m_H(N) \leq N + 1 \)

\( H \) is positive intervals \( m_H(N) \leq \frac{1}{2} N^2 + \frac{1}{2} N + 1 \)

\( H \) is 2-D perceptron \( m_H(N) \leq \frac{1}{6} N^3 + \frac{5}{6} N + 1 \)

\( H \) is the convex sets \( m_H(N) = 2^N \)

• Introduced the VC dimension

The single parameter that characterizes the growth function:

\[
m_H(N) \leq \sum_{i=0}^{k-1} \binom{N}{i} = \sum_{i=0}^{d_{vc}} \binom{N}{i}
\]

max power is \( N^{d_{vc}} \)

\[
m_H(N) \leq N^{d_{vc}} + 1
\]
Review of Lecture 7

• Implications to learning

\[ E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{1}{2N} \log \frac{2|H|}{\delta}} \] w.p. at least \( 1 - \delta \)

\[ E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \log \frac{4m_H(2N)}{\delta}} \]

and bounding \( m_H(2N) \) by the VC dimension, we get

\[ \epsilon = \sqrt{\frac{8}{N} \ln \frac{4((2N)^{d_{vc}} + 1)}{\delta}} \]

\( d_{vc}(H) \) is finite \( \Rightarrow g \in H \) will generalize.

(with enough data)

• How much data?

Set the error bar at \( \epsilon \).

\[ \epsilon = \sqrt{\frac{8}{N} \ln \frac{4((2N)^{d_{vc}} + 1)}{\delta}} \]

Solve for \( N \).

\[ N = \frac{8}{\epsilon^2 \ln \frac{4((2N)^{d_{vc}} + 1)}{\delta}} = O(d_{vc} \ln N) \]

Practical rule-of-thumb: \( N \geq 10d_{vc} \)

Not so bad! And remember, \( d_{vc} \) may be \( << \) than the number of parameters in your model...

\[ x \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow y \]

..., since it measures the effective number of parameters.
Today’s Lecture

• A whirlwind tour of the practical issues with training deep neural networks

• Improving generalization through practical techniques

(Many slides adapted from Yaser Abu-Mostafa and Malik Magdon-Ismail, with permission of the authors. Thanks guys!)
The learning rate

“The learning rate is perhaps the most important hyper parameter. If you have time to tune only one hyper parameter, tune the learning rate.”

—DLB, Chapter 11, p. 424

• Controls the **effective capacity** of the model
  - Highest when learning rate is correct
  - Lower when it is especially large or small

• Has a u-shaped curve for training error
  - **Too large**: GD can increase rather than decrease the training error.
    (idealized case: happens when >2X optimal value)
  - **Too small**: Training not only slower but can get permanently stuck (A mystery!)
The learning rate

• Loss during training is also affected:
  - **Low learning rates:** improvements are linear
  - **High learning rates:** look exponential but get stuck

• A typical loss function over time (CIFAR-10):
  - Slightly too low?
  - **Batch size** a bit too small (Too noisy?)

(Credit: Stanford CS231N)
Selecting a learning rate

- Standard practice is to perform a search to select it.
- The learning rate you select will depend on
  1. The model (number of params, arch)
  2. The optimization algorithm
  3. The batch or mini-batch size
  4. What you empirically observe

(image credit: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/)
Batch and mini-batch algorithms

- Objective function (loss function) usually decomposes as a sum over training examples
- Optimization for machine learning:
  - Typically compute parameter updates based on expected value of loss function

For maximum likelihood estimation:

\[ W_{ML} = \arg \max_W \sum_{i=1}^{N} \ln P(x_i, y_i; W) \]

which is equivalent to maximizing:

\[ J(W) = \mathbb{E}_{x,y \sim D} \ln P(x, y; W) \]

where \( J(W) \) is the loss function.
Batch and mini-batch algorithms

The gradient of the loss is:

\[ \nabla_W J(W) = \mathbb{E}_{x,y \sim D} \nabla_W \ln P(x, y; W) \]

- Computing exactly is very expensive since we need to evaluate on every data point in dataset
- Better to randomly sample a small number of examples and average over only these. Why?
  - Standard error of mean estimated from \( n \) samples is \( \sigma/\sqrt{n} \).
  - Less than linear returns for using more examples to estimate the gradient.

100 examples/update \( \rightarrow \) 100X more computation \( \rightarrow \) 10X reduction in SE \( \rightarrow \) 10,000 examples/update

Most optimization algorithms converge faster (in terms of total computation, not updates) if allowed to rapidly compute approximate estimates of gradient.
Mini-batch size and learning curve

• Gradient descent is smooth since updates the gradient once per full pass through the dataset.

• SGD is the other extreme, with $N$ gradient updates per pass through the dataset.
  - Curve is erratic since it is not minimizing total error at each iteration, but rather, error on a specific data point.

• Most algorithms for deep learning fall in between, using more than 1, but fewer than all of the training examples.

(500 training examples, 2-layer network, 5 hidden units, learning rate $\eta = 0.01$)
Other arguments for mini-batching

- Potentially reduces redundant computation
  - Worst case: all samples in the dataset are identical copies of each other
  - Could compute the correct gradient with 1 example
    - $N$ times less than a naive approach
  - In practice, may find large number of examples that all make similar contributions
What to consider when selecting a mini-batch size

Larger batch sizes provide a more accurate gradient estimate, but with sub-linear returns.

Minimum batch size driven by multi-core architectures (underutilized by extremely small batches)

Amount of memory scales with batch size for parallel computing (limiting factor for size)

Some hardware offers better runtimes with specific array sizes (e.g., powers of 2)

Small batches can offering a regularizing effect, perhaps due to the noise they add to the learning process. Size of 1 may offer best generalization error.

Small batches may require a small learning rate to maintain stability because of the high variance in the gradient estimate. As a result, total runtime may be very high!

(See DLB, Ch 8.1.3)
Selecting mini-batches at random

• Crucial that mini-batches be selected randomly
  - Computing an unbiased estimate of the gradient requires samples to be independent
  - Also desire subsequent gradient estimates to be independent from each other
  - Many datasets are naturally organized such that successive examples are highly correlated. For example:
    1. Blood samples taken from one patient, then the next (5 times over)
    2. Housing prices organized by zip code
    3. Engine measurements organized by serial number (order off assembly line)

• Shuffle the dataset (at least once!)
Choosing the initial weights can be tricky.

Example:

Initialize the weights so that \( \tanh(w^T x_n) \approx \pm 1 \).

What happens?

**Gradient is close to zero and algorithm doesn’t make progress!**
Initialization Strategies

Simple solutions:

1. Initialize the weights to **small random values**, where \( \tanh(w^T x_n) \approx 0 \)

   Provides algorithm with flexibility to adjust the weights to fit the data.

2. Initialize using **Gaussian random weights**, \( w_i \sim \mathcal{N}(0, \sigma^2_w) \), where \( \sigma^2_w \) is small. How small?

   Want \( |w^T x_n|^2 \) to be small.

   Since \( \mathbb{E} \left[ |w^T x_n|^2 \right] = \sigma^2_w \| x_n \|^2 \), we should choose \( \sigma^2_w \) so that \( \sigma^2_w \cdot \max_n \| x_n \|^2 \ll 1 \).

What can go wrong if we initialize all the weights to exactly zero?
Termination

When do we stop training a deep learning model?

Risky to rely solely on the magnitude of the gradient to stop.

- May stop prematurely in a relatively flat region.

May combat this by using a combination of stopping criteria:

1. Small improvement in error
2. Small error
3. Upper bound on iterations
Termination

Removing the non-linearity from the output layer can help as well.

Turn a classification problem into a regression problem for training purposes:
- Fit the classification data $y_n = \pm 1$ as if it were a regression problem
- Use the identity function as output node transformation instead of $\tanh(\cdot)$

Can greatly help combat the ‘flat regions’ during training:
- Avoids exceptionally flat nature of $\tanh(\cdot)$ when its argument gets large.
- If the weights get large early in training, surface starts to look flat. No progress!
- Can recover from an initial bad move if it happens to take you to large weights.
  (linear output never saturates)
Early stopping

Terminating training early may have its benefits.

An iterative method like GD does not explore the full hypothesis set all at once.

\[ \mathcal{H}_1 = \{ w : \| w - w_0 \| \leq \eta \} \]

Continuing...

\[ \mathcal{H}_2 = \mathcal{H}_1 \cup \{ w : \| w - w_1 \| \leq \eta \} \]

\[ \mathcal{H}_3 = \mathcal{H}_2 \cup \{ w : \| w - w_2 \| \leq \eta \} \]

\[ H_1 \subset H_2 \subset H_3 \subset H_4 \subset \ldots \]
Early stopping

As $t$ increases, $E_{in}(w_t)$ is decreasing, and $d_{vc}(H_t)$ is increasing.

Expect to see an approximation-generalization trade-off:

What is omega in this case?

Suggests is may be better to stop at some $t^*$, well before reaching minimum of $E_{in}$. 
Early stopping

Reinforces our theory:

- Test error **initially decreases** as approximation gain overcomes the worsening generalization error.
- Test error **eventually increases** as generalization error begins to dominate.

**Overfitting!**

How do we determine $t^*$?
Early stopping example: Digits data

How do we determine $t^*$?
Use a ‘validation set’. Output $w^*$. And don’t add back in ALL of the data and retrain!!
A trade-off exists in choosing the size of the validation set:

- Too large and little data to train on.
- Too small and the validation error will not be reliable.

Rule of thumb is to set aside 10-20%.
Overfitting, really?!

Example was a small network, with a relatively large dataset.

Why would overfitting be a problem?

- The data is noisy
- The target function is complex

Therefore, both stochastic and deterministic noise are significant.

- Better to stop early at $t^*$ and constrain the learning to $H_{t^*}$.
- A form of regularization (Next time!)

Zero!
Further reading


