Neural Network Training: The Gory Details

(Or, how to be a helicopter parent to a neural network)

(Or, why AI is not about to be solved any time soon)
Outline

• Optimization
  • Mini-batch SGD
  • Learning rate decay
  • Adaptive methods

• Massaging the numbers
  • Data augmentation
  • Data preprocessing
  • Weight initialization
  • Batch normalization

• Regularization
  • Classic regularization: L2 and L1
  • Dropout
  • Label smoothing

• Test time: ensembles, averaging predictions
An in-depth look into optimization

Caspar David Friedrich, *Wanderer Above a Sea of Fog*, 1818
Mini-batch SGD

- Iterate over epochs
  - Iterate over dataset mini-batches \((x_1, y_1), \ldots, (x_b, y_b)\)
  - Compute gradient of the mini-batch loss:
    \[
    \nabla \hat{L} = \frac{1}{b} \sum_{i=1}^{b} \nabla l(w, x_i, y_i)
    \]
  - Update parameters:
    \[
    w \leftarrow w - \eta \nabla \hat{L}
    \]
  - Check for convergence, decide whether to decay learning rate

- What are the hyperparameters?
  - Mini-batch size, learning rate decay schedule, deciding when to stop
SGD and mini-batch size

- Larger mini-batches: more expensive and less frequent updates, lower gradient variance, more parallelizable
- In the literature, SGD with larger batches is generally reported to generalize more poorly (e.g., Keskar et al., 2016)
  - But can be made to work by using larger learning rates with larger mini-batches (Goyal et al., 2017)
Learning rate decay

- **Exponential decay**: $\eta = \eta_0 e^{-kt}$, where $\eta_0$ and $k$ are hyperparameters, $t$ is the iteration or epoch number.

- **$1/t$ decay**: $\eta = \eta_0/(1 + kt)$

- **Step decay**: reduce rate by a constant factor every few epochs, e.g., by 0.5 every 5 epochs, 0.1 every 20 epochs.

- **Manual**: watch validation error and reduce learning rate whenever it stops improving.
Diagnosing learning rates

![Image source: Stanford CS231n]
A typical phenomenon

Why does the learning curve look like this?

Image source: Stanford CS231n
A typical phenomenon

Possible explanation

Corresponding Learning Curve
Debugging learning curves

- Not training: Bug in update calculation?
- Error increasing: Bug in update calculation?
- Error decreasing: Not converged yet
- Slow start: Suboptimal initialization?
- Possible overfitting
- Definite overfitting

Image source: Stanford CS231n
Early stopping

- Idea: do not train a network to achieve too low training error
- Monitor validation error to decide when to stop

Figure from Deep Learning Book
Advanced optimizers

- SGD with momentum
- RSMProp
- Adam
SGD with momentum

What will SGD do?
SGD with momentum

- Introduce a “momentum” variable $m$ and associated “friction” coefficient $\beta$:
  \[
  m \leftarrow \beta m - \eta \nabla L
  \]
  \[
  w \leftarrow w + m
  \]
- Typically start with $\beta = 0.5$, gradually increase over time
SGD with momentum

- Introduce a “momentum” variable $m$ and associated “friction” coefficient $\beta$:
  \[
  m \leftarrow \beta m - \eta \nabla L \\
  w \leftarrow w + m
  \]
- Move faster in directions with consistent gradient
- Avoid oscillating in directions with large but inconsistent gradients
SGD with momentum

- Introduce a “momentum” variable $m$ and associated “friction” coefficient $\beta$:
  $$m \leftarrow \beta m - \eta \nabla L$$
  $$w \leftarrow w + m$$

- Nesterov momentum: evaluate gradient at “lookahead” position $w + \beta m$
Adaptive per-parameter learning rates

- Gradients of different layers have different magnitudes
- Want an automatic way to set different learning rates for different parameters
Adagrad

• Keep track of history of gradient magnitudes, scale the learning rate for each parameter based on this history:

\[
\begin{align*}
\nu_k &\leftarrow \nu_k + \left\| \frac{\partial L}{\partial w_k} \right\|^2 \\
w_k &\leftarrow w_k - \frac{\eta}{\sqrt{\nu_k + \epsilon}} \frac{\partial L}{\partial w_k}
\end{align*}
\]

• Parameters with small gradients get large updates and vice versa

• Long-ago gradient magnitudes are not “forgotten” so learning rate decays too quickly

J. Duchi, Adaptive subgradient methods for online learning and stochastic optimization, JMLR 2011
RMSProp

• Introduce decay factor $\beta$ (typically $\geq 0.9$) to downweight past history exponentially:

\[ v_k \leftarrow \beta v_k + (1 - \beta) \left\| \frac{\partial L}{\partial w_k} \right\|^2 \]

\[ w_k \leftarrow w_k - \frac{\eta}{\sqrt{v_k} + \epsilon} \frac{\partial L}{\partial w_k} \]

Adam

- Combine RMSProp with momentum:

\[
m \leftarrow \beta_1 m + (1 - \beta_1) \nabla L
\]

\[
v_k \leftarrow \beta v_k + (1 - \beta) \left\| \frac{\partial L}{\partial w_k} \right\|^2
\]

\[
w_k \leftarrow w_k - \frac{\eta}{\sqrt{v_k} + \epsilon} m_k
\]

- Default parameters from paper:

\[
\beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 1e - 8
\]

- Full algorithm includes bias correction term to account for \( m \) and \( v \) starting at 0:

\[
\hat{m} = \frac{m}{1 - \beta_1^t}, \quad \hat{v} = \frac{v}{1 - \beta_2^t} (t \text{ is the timestep})
\]

D. Kingma and J. Ba, Adam: A method for stochastic optimization, ICLR 2015
Which optimizer to use in practice?

- Adaptive methods tend to reduce initial training error faster than SGD
  - Adam with default parameters is a popular choice, SGD+momentum may work better but requires more tuning
- However, adaptive methods may quickly plateau on the validation set or generalize more poorly
  - Use Adam first, then switch to SGD?
  - Or just stick with plain old SGD? (Wilson et al., 2017)
- All methods require careful tuning and learning rate control
Other optimization tricks

- **Adding noise to gradients**: SGD with Langevin dynamics helps to jump out of local minima (e.g., Welling and Teh, 2011)

- **Cyclical learning rates**: increase learning rate from time to time to jump out of local minima (Smith 2017)
Massaging the numbers
Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
Data augmentation

• Introduce transformations not adequately sampled in the training data
  • Geometric: flipping, rotation, shearing, multiple crops
  • Photometric: color transformations
Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
  - Photometric: color transformations
  - Other: add noise, compression artifacts, lens distortions, etc.
Data augmentation

- Introduce transformations not adequately sampled in the training data
- Limited only by your imagination and time/memory constraints!
- Avoid introducing obvious artifacts
Data preprocessing

• Zero centering
  • Subtract *mean image* – all input images need to have the same resolution
  • Subtract *per-channel means* – images don’t need to have the same resolution

• Optional: rescaling – divide each value by (per-pixel or per-channel) standard deviation

• Be sure to apply the same transformation at training and test time!
  • Save training set statistics and apply to test data
Weight initialization

- What’s wrong with initializing all weights to the same number (e.g., zero)?
Weight initialization

- Typically: initialize to random values sampled from zero-mean Gaussian: $w \sim \mathcal{N}(0, \sigma^2)$
- Standard deviation matters!
- Key idea: avoid reducing or amplifying the variance of layer responses, which would lead to vanishing or exploding gradients

- Common heuristics:
  - $\sigma = 1/\sqrt{n_{\text{in}}}$, where $n_{\text{in}}$ is the number of inputs to a layer
  - $\sigma = 2/\sqrt{n_{\text{in}} + n_{\text{out}}}$ (Glorot and Bengio, 2010)
  - $\sigma = \sqrt{2/n_{\text{in}}}$ for ReLU (He et al., 2015)

- Initializing biases: just set them to 0

Batch normalization

- The authors’ intuition

Batch normalization

- Key idea: data shifting/rescaling is a differentiable operation, so the network can learn how best to normalize the data
- Statistics of activations (outputs) from a given layer across the dataset can be approximated by statistics from a mini-batch

Batch normalization

Input: Values of $x$ over a mini-batch: $B = \{x_1...m\}$;
Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

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

Why?

Batch normalization

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

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

\[
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)
\end{align*}
\]

**At test time:**

// mini-batch mean training set

// mini-batch variance training set

// normalize

// scale and shift

Batch normalization

- Insert BN layers right after conv or FC layers, before ReLU nonlinearity

Batch normalization

• Benefits
  • Prevents exploding and vanishing gradients
  • Keeps most activations away from saturation regions of non-linearities
  • Accelerates convergence of training
  • Makes training more robust w.r.t. hyperparameter choice, initialization

• Pitfalls
  • Behavior depends on composition of minibatches, can lead to hard-to-catch bugs if there is a mismatch between training and test regime (example)

Why does BatchNorm *really* work?

- It may have to do not with internal covariate shift (ICS), but with making the optimization problem much smoother (Santurkar et al., 2018)
- *Is ICS even a thing?* (Lipton and Steinhardt, 2018)
Regularization

- Techniques for controlling the capacity of a neural network to prevent overfitting
Review: L2 regularization

• Regularized objective:

\[ \hat{L}(w) = \frac{\lambda}{2} \|w\|_2^2 + \sum_{i=1}^{n} l(w, x_i, y_i) \]

• Gradient of objective:

\[ \nabla \hat{L}(w) = \lambda w + \sum_{i=1}^{n} \nabla l(w, x_i, y_i) \]

• SGD update:

\[
\begin{align*}
    w &\leftarrow w - \eta (\lambda w + \nabla l(w, x_i, y_i)) \\
    w &\leftarrow (1 - \eta \lambda)w - \eta \nabla l(w, x_i, y_i)
\end{align*}
\]

• Interpretation: weight decay
L1 regularization

- Regularized objective:

\[ \hat{L}(w) = \lambda \|w\|_1 + \sum_{i=1}^{n} l(w, x_i, y_i) \]

\[ = \lambda \sum_{d} |w_d| + \sum_{i=1}^{n} l(w, x_i, y_i) \]

- Gradient: \( \nabla \hat{L}(w) = \lambda \text{sgn}(w) + \sum_{i=1}^{n} \nabla l(w, x_i, y_i) \)

- SGD update:

\[ w \leftarrow w - \eta \lambda \text{sgn}(w) - \eta \nabla l(w, x_i, y_i) \]

- Interpretation: encouraging sparsity
Other types of regularization

- Adding noise to the inputs
  - Recall motivation of max margin criterion
  - In simple scenario (linear model, quadratic loss, Gaussian noise), this is equivalent to weight decay
  - Data augmentation is a more general form of this
- Adding noise to the weights
Dropout

- At training time, in each forward pass, turn off some neurons with probability $p$
- At test time, to have deterministic behavior, multiply output of neuron by $p$

 Dropout

• Intuitions
  • Prevent “co-adaptation” of units, increase robustness to noise
  • Train *implicit ensemble*

Current status of dropout

- Against
  - Slows down convergence
  - Made redundant by batch normalization or possibly even **clashes with it**
  - Unnecessary for larger datasets or with sufficient data augmentation

- In favor
  - Can still help in certain scenarios: e.g., used in Wide Residual Networks
Label smoothing

• **Idea**: avoid overly confident predictions, account for label noise

• When using softmax loss, replace hard 1 and 0 prediction targets with “soft” targets of $1 - \epsilon$ and $\frac{\epsilon}{C-1}$

• Used in Inception-v2 architecture
Test time

• **Ensembles:** train multiple independent models, then average their predicted label distributions
  • Gives 1-2% improvement in most cases
  • Can take multiple snapshots of models obtained during training, especially if you cycle the learning rate

G. Huang et al., *Snapshot ensembles: Train 1, get M for free*, ICLR 2017
Test time

- Average predictions across multiple crops of test image
  - There is a more elegant way to do this with *fully convolutional networks* (FCNs)
Attempt at a conclusion

• Training neural networks is still a black art
• Process requires close “babysitting”
• For many techniques, the reasons why, when, and whether they work are in active dispute
• Read everything but don’t trust anything
• It all comes down to (principled) trial and error

Salvator Rosa, *Witches at Their Incantations*, 1646