Neural network training: Beyond the basics
Outline

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

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

• Regularization
  • Dropout
  • Label smoothing

• Test time: ensembles, averaging predictions

• Transfer learning, distillation
An overview of optimization techniques

Caspar David Friedrich, Wanderer above a sea of fog, 1817
Mini-batch SGD

- Iterate over epochs
  - Group data into mini-batches of size $b$
    - Compute gradient of the loss for the mini-batch $(x_1, y_1), ..., (x_b, y_b)$:
      \[
      \nabla \hat{L} = \frac{1}{b} \sum_{i=1}^{b} \nabla l(w, x_i, y_i)
      \]
      
    - 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
Setting the mini-batch size

- Larger mini-batches: more expensive and less frequent updates, lower gradient variance, more parallelizable
- SGD with larger batches may generalize more poorly (e.g., Keskar et al., 2017)
- But can be made to work well by carefully controlling learning rate and addressing other optimization issues (Goyal et al., 2018)
Setting the learning rate

Source: Stanford CS231n

Figure source
Learning rate decay

- **Decay formulas**
  - Exponential: $\eta_t = \eta_0 e^{-kt}$, where $\eta_0$ and $k$ are hyperparameters, $t$ is the iteration or epoch number
  - Inverse: $\eta_t = \eta_0/(1 + kt)$
  - Inverse sqrt: $\eta_t = \eta_0/\sqrt{t}$
  - Linear: $\eta_t = \eta_0(1 - t/T)$, where $T$ is the total number of epochs
  - Cosine: $\eta_t = \frac{1}{2}\eta_0(1 + \cos(t\pi/T))$
Learning rate decay

- Decay formulas
- Most common in practice:
  - **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
    - “Patience” hyperparameter: number of epochs without improvement before reducing learning rate
A typical phenomenon

Possible explanation

Image source: Stanford CS231n
Learning rate decay

• Decay formulas
• Most common in practice:
  • **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
    • “Patience” hyperparameter: number of epochs without improvement before reducing learning rate
  • **Warmup**: train with a low learning rate for a first few epochs, or linearly increase learning rate before transitioning to normal decay schedule ([Goyal et al.](https://arxiv.org/abs/1706.09879), 2018)
Diagnosing learning curves: Obvious problems

- Not training
  - Bug in update calculation?

- Error increasing
  - Bug in update calculation?

- Get NaNs in the loss after a number of iterations:
  - Numerical instability

- Weird cyclical patterns in loss:
  - Data not shuffled

Source: Stanford CS231n
Diagnosing learning curves: Subtler behaviors

- Not converged yet: Keep training, possibly increase learning rate
- Slow start: Bad initialization?
- Possible overfitting
- Definite overfitting

Source: Stanford CS231n
When to stop training?

- Monitor validation error to decide when to stop
  - “Patience” hyperparameter: number of epochs without improvement before stopping
  - *Early stopping* can be viewed as a kind of regularization

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 \( w + \beta m \)
### Adagrad: Adaptive per-parameter learning rates

- Keep track of history of gradient magnitudes, scale the learning rate for each parameter based on this history.
- For each dimension $k$ of the weight vector:

  $\nu^{(k)} \leftarrow \nu^{(k)} + \left( \frac{\partial L}{\partial w^{(k)}} \right)^2$

  $w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{\nu^{(k)}} + \epsilon} \frac{\partial L}{\partial w^{(k)}}$

  Update running sum of squared magnitudes of gradient w.r.t. $k$th weight.
  Scale learning rate for $k$th weight by inverse of the magnitude, update $k$th weight.

- Parameters with small gradients get large updates and vice versa.
- Problem: 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:

$$
\nu^{(k)} \leftarrow \beta \nu^{(k)} + (1 - \beta) \left( \frac{\partial L}{\partial w^{(k)}} \right)^2
$$

$$
w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{\nu^{(k)}} + \epsilon} \frac{\partial L}{\partial w^{(k)}}
$$

Adam: Combine RMSProp with momentum

• Update momentum:

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

• For each dimension \( k \) of the weight vector:

\[ \nu^{(k)} \leftarrow \beta_2 \nu^{(k)} + (1 - \beta_2) \left( \frac{\partial L}{\partial w^{(k)}} \right)^2 \]

\[ w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{\nu^{(k)}} + \epsilon} m^{(k)} \]

• Full algorithm includes *bias correction* to account for \( m \) and \( \nu \) starting at 0: \( \hat{m} = \frac{m}{1 - \beta_1^t}, \hat{\nu} = \frac{\nu}{1 - \beta_2^t} \) (\( t \) is the timestep)

• Default parameters from paper are reputed to work well for many models: \( \beta_1 = 0.9, \beta_2 = 0.999, \eta = 1e - 3, \epsilon = 1e - 8 \)
  
Which optimizer to use in practice?

- Adaptive methods tend to reduce initial training error faster than SGD and are “safer”
  - **Andrej Karpathy**: “In the early stages of setting baselines I like to use Adam with a learning rate of 3e-4. In my experience Adam is much more forgiving to hyperparameters, including a bad learning rate. For ConvNets a well-tuned SGD will almost always slightly outperform Adam, but the optimal learning rate region is much more narrow and problem-specific.”
- Use Adam at first, then switch to SGD?
- However, some literature reports problems with adaptive methods, such as failing to converge or generalizing poorly ([Wilson et al. 2017](https://arxiv.org/abs/1705.08741), [Reddi et al. 2018](https://arxiv.org/abs/1811.00271))
  - YMMV!
Neural network training: Beyond the basics

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive methods: SGD with momentum, RMSProp, Adam

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

- Regularization
  - Dropout
  - Label smoothing

- Test time: ensembles, averaging predictions
- Transfer learning, distillation
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

Image source
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 artifacts
Data augmentation

• Introduce transformations not adequately sampled in the training data
• Limited only by your imagination and time/memory constraints!
• Avoid introducing artifacts
• Automatic augmentation strategies: AutoAugment, RandAugment
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:
  - Xavier initialization: $\sigma^2 = 1/n_{\text{in}}$ or $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$, where $n_{\text{in}}$ and $n_{\text{out}}$ are the numbers of inputs and outputs to a layer (Glorot and Bengio, 2010)
  - Kaiming initialization (for ReLU): $\sigma^2 = 2/n_{\text{in}}$ (He et al., 2015)
  - Initializing biases: just set them to 0

Batch normalization

- The authors’ intuition

Batch normalization

- **Key idea**: shifting and rescaling are differentiable operations, so the network can *learn* how to best 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: $\mathcal{B} = \{x_1, \ldots, x_m\}$; Parameters to be learned: $\gamma, \beta$

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

\[
\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
\]

\[
\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad \text{// mini-batch variance}
\]

\[
\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{// normalize}
\]

\[
y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad \text{// scale and shift}
\]

**Why?**

Batch normalization

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

**Output:** $\{y_i = BN_{\gamma,\beta}(x_i)\}$

At test time (usually):

$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$  // mini-batch mean

$\sigma^2_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$  // mini-batch variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \epsilon}}$  // normalize

$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$  // scale and shift

Batch normalization

- Common configuration: insert BN layers right after conv or FC layers, before ReLU nonlinearity (but this is purely empirical)

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 mini-batches, can lead to hard-to-catch bugs if there is a mismatch between training and test regime (example)
  • Doesn’t work well for small mini-batch sizes
  • Cannot be used in recurrent models
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](https://arxiv.org/pdf/1803.08820.pdf))
Other types of normalization

- **Layer normalization** (Ba et al., 2016)
- **Instance normalization** (Ulyanov et al., 2017)
- **Group normalization** (Wu and He, 2018)
- **Weight normalization** (Salimans et al., 2016)

Y. Wu and K. He, *Group Normalization*, ECCV 2018
Outline

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Adaptive methods
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization
- Regularization
  - Dropout
  - Label smoothing
Recall: Regularization

- Techniques for controlling the capacity of a neural network to prevent overfitting
- Classic regularization: L1, L2
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: A Simple Way to Prevent Neural Networks from Overfitting. JMLR 2014
Dropout

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

**Dropout: A Simple Way to Prevent Neural Networks from Overfitting.** JMLR 2014
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 for certain models and in certain situations: 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](https://www.tensorflow.org/api_docs/python/tf/train/InceptionV3) architecture
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
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 (increase to jump out of local minima)

G. Huang et al., [Snapshot ensembles: Train 1, get M for free](https://arxiv.org/abs/1606.04272), 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)
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

• Transfer learning, distillation
How to use a pre-trained network for a new task?

• Strategy 1: Use as feature extractor

Remove these layers

Use as off-the-shelf feature
How to use a pre-trained network for a new task?

- Strategy 2: Transfer learning

Train new prediction layer(s)

Keep frozen or fine-tune

VGG16
Distillation

1. Train a *teacher* network on initial labeled dataset
2. Save the softmax outputs the teacher network for each training example
3. Train a *student* network with cross-entropy loss using the softmax outputs of the teacher network as targets

- Many uses
  - Compressing a larger model (or even an ensemble) into a smaller one
  - “Copying” a black-box teacher model (e.g., network you can only access via an API)
  - Extending a network to additional tasks without “forgetting” old tasks ([Li and Hoiem](https://example.com), 2017)

Some take-aways

- 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
- Further reading: A. Karpathy, A recipe for training neural networks