Training and optimization
Stochastic gradient descent

• Gradient on single example = unbiased sample of true gradient
• Idea: at each iteration sample single example $x^{(t)}$

$\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x^{(t)}), y^{(t)})$

step size

• Con: variance in estimate of gradient $\rightarrow$ slow convergence, jumping near optimum
Minibatch stochastic gradient descent

- Compute gradient on a small batch of examples
- Same mean (=true gradient), but variance inversely proportional to minibatch size

\[
\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \frac{1}{|B(t)|} \sum_{(x,y) \in B(t)} \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x), y)
\]
Momentum

• *Average* multiple gradient steps
• Use *exponential averaging*

\[
\mathbf{p}^{(t+1)} \leftarrow (1 - \mu)\mathbf{p}^{(t)} - \mu \frac{1}{|B^{(t)}|} \sum_{(x,y) \in B^{(t)}} \nabla_{\mathbf{w}} L(h_{\mathbf{w}}(x), y)
\]

\[
\mathbf{w}^{(t+1)} \leftarrow \mathbf{w}^{(t)} - \lambda \mathbf{p}^{(t+1)}
\]
Weight decay

• Add \(-aw(t)\) to the gradient
• Prevents \(w(t)\) from growing to infinity
• Equivalent to L2 regularization of weights
Learning rate decay

• Large step size / learning rate
  • Faster convergence initially
  • Bouncing around at the end because of noisy gradients

• Learning rate must be decreased over time

• Usually done in steps
Convolutional network training

- Initialize network
- Sample *minibatch* of images
- Forward pass to compute loss
- Backpropagate loss to compute gradient
- Combine gradient with momentum and weight decay
- Take step according to current learning rate
Beyond sequences: computation graphs

• Multi-layer perceptrons and first convolutional networks were *sequences of functions*

• In general, can have *arbitrary DAGs* of functions
Computation graphs

• Each node implements two methods
  • A “forward”
    • Computes output given input
  • A “backward”
    • Computes derivative of z w.r.t input, given derivative of z w.r.t output
Computation graphs

\[ f_i(a, b, c) \rightarrow d \]
Computation graphs
Exploring convnet architectures
Deeper is better

Challenge winner's accuracy

- 2010: 28 layers
- 2011: 25 layers
- 2012: 16 layers
- 2013: 10 layers
- 2014: 5 layers

Layers increase over time, indicating better accuracy.
Deeper is better

Challenge winner's accuracy

- 2010: Alexnet
- 2011: Alexnet
- 2012: Alexnet
- 2013: VGG16
- 2014: VGG16
The VGG pattern

• Every convolution is 3x3, padded by 1
• Every convolution followed by ReLU
• ConvNet is divided into “stages”
  • Layers within a stage: no subsampling
  • Subsampling by 2 at the end of each stage
• Layers within stage have same number of channels
• Every subsampling → double the number of channels
Challenges in training: exploding / vanishing gradients

• Vanishing / exploding gradients

\[
\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}
\]

• If each term is (much) greater than 1 \(\rightarrow\) explosion of gradients
• If each term is (much) less than 1 \(\rightarrow\) vanishing gradients
Challenges in training: dependence on init
Solutions

• Careful init

• Batch normalization

• Residual connections
Careful initialization

• Key idea: want variance to remain approx. constant
  • Variance increases in backward pass => exploding gradient
  • Variance decreases in backward pass => vanishing gradient

• “MSRA initialization”
  • weights = Gaussian with 0 mean and variance = 2/(k*k*d)
Batch normalization

• Key idea: normalize so that each layer output has zero mean and unit variance
  • Compute mean and variance for each channel
  • Aggregate over batch
  • Subtract mean, divide by std

• Need to reconcile train and test
  • No "batches" during test
  • After training, compute means and variances on train set and store

Residual connections

- In general, gradients tend to vanish
- Key idea: allow gradients to flow unimpeded

\[
\frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial f_{i+1}(z_i, w_{i+1})}{\partial z_i} = \frac{\partial f_{i+1}(z_i, w_{i+1})}{\partial z_i}
\]

\[
\frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}
\]
Residual connections

• In general, gradients tend to vanish
• Key idea: allow gradients to flow unimpeded

\[ z_{i+1} = g_{i+1}(z_i, w_{i+1}) + z_i \]

\[ \frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial g_{i+1}(z_i, w_{i+1})}{\partial z_i} + I \]

\[ \frac{\partial z}{\partial z_i} = \frac{\partial z}{\partial z_{n-1}} \frac{\partial z_{n-1}}{\partial z_{n-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i} \]
Residual connections

- Assumes all $z_i$ have the same size
- True within a stage
- Across stages?
  - Doubling of feature channels
  - Subsampling
- Increase channels by 1x1 convolution
- Decrease spatial resolution by subsampling

\[ z_{i+1} = g_{i+1}(z_i, w_{i+1}) + \text{subsample}(Wz_i) \]
A residual block

• Instead of single layers, have residual connections over block
Bottleneck blocks

• Problem: When channels increases, 3x3 convolutions introduce many parameters
  • $3 \times 3 \times c^2$

• Key idea: use 1x1 to project to lower dimensionality, do convolution, then come back
  • $c \times d + 3 \times 3 \times d^2 + d \times c$
The ResNet pattern

• Decrease resolution substantially in first layer
  • Reduces memory consumption due to intermediate outputs

• Divide into stages
  • maintain resolution, channels in each stage
  • halve resolution, double channels between stages

• Divide each stage into residual blocks

• At the end, compute average value of each channel to feed linear classifier
Putting it all together - Residual networks

Challenge winner's accuracy
Computational complexity
Analyzing computational complexity

• What is the computational complexity of a single convolutional layer?
  • $h \times w \times c$ input and output
  • $k \times k$ kernel

• Space:
  • Input/output: $hwc$
  • Filters: $k^2c^2$

• Time (Flops): $hwk^2c^2$
Reducing computational complexity

• ...while maintaining accuracy?
• Multiple ways:
  • Make architecture \textit{a priori} cheaper
  • Make \textit{weights} and \textit{operations} cheaper
  • Make inference adaptive
Cheaper convolutional blocks

- **Standard convolution:**
  - Each filter operates on all channels
  - Single $k \times k$ filter operating on $c$ channels producing one output channel: $k^2c$ parameters, cost
  - $c$ such filters: $k^2c^2$ parameters, cost

- **Depthwise separable convolution**
  - Each filter operates on a single channel
  - $c$ filters operating on $c$ channels: $k^2c$ parameters, cost
  - But each channel is independently processed
  - Add a 1x1 convolution at the end with cost $c^2 : k^2c + c^2$ parameters
Cheaper convolutional blocks

• Depthwise separable convolutions are specific instance of more general idea: *grouped convolutions*
• Grouped convolutions in original AlexNet network
• Grouped convolution:
  • Divide input channels into $g$ groups
  • Apply convolutional layers on each group independently
  • Concatenate
Grouped and depth-wise convolutions


Table 4. Depthwise Separable vs Full Convolution MobileNet

<table>
<thead>
<tr>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv MobileNet</td>
<td>71.7%</td>
<td>4866</td>
<td>29.3</td>
</tr>
<tr>
<td>MobileNet</td>
<td>70.6%</td>
<td>569</td>
<td>4.2</td>
</tr>
</tbody>
</table>
Other architectural changes

• Biggest memory consumption: large feature maps
Other architectural changes

• Biggest memory consumption: large feature maps

• Simple solution: reduce resolution early
Other architectural changes

• Biggest memory consumption: large feature maps

• Simple solution (ResNet):
  • Reduce resolution drastically (\(\times 4\)) early

• More sophisticated changes: Inverted residuals (MobileNet v2)

Other kinds of connections

• DenseNets
  • Replace addition of residuals with concatenation
  • Alternative to solving vanishing gradient problem
  • Should *increase* number of parameters, but *decreases* them
  • Better re-use of features

*Figure 1*: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.
Adaptive inference

• Some examples are harder than others
• Should be able to use different amounts of computation for different examples
• Version 1: skip some residuals


Adaptive inference

• Some examples are harder than others
• Should be able to use different amounts of computation for different examples
• Version 1: skip some residuals


Adaptive inference

• Some examples are harder than others
• Should be able to use different amounts of computation for different examples
• Version 2: reduce resolution at different rates

Huang, Gao, et al. "Multi-scale dense networks for resource efficient image classification." ICLR 2018
Compressing model weights

• All of model storage: filters
• Flops also scale with non-zero entries in filters (in principle)
• Compress filters
  • Sparsify them
  • Represent them with fewer bits
Pruning network connections

- Simple approach: prune weights that are below a threshold
- Retrain rest of the weights
- Repeat

Pruning network connections

• Simple approach: prune weights that are below a threshold
• Retrain rest of the weights
• Repeat

• Sophisticated alternative
  • Train with *regularizer* that penalizes expensive connections
  • Prune
  • If model within budget, expand and retrain

Filter quantization

• Two questions:
  • How do we quantize?
  • Quantization → discrete values. How do we optimize?

• Example 1: cluster
  • Weights → indices into dictionary
  • Update dictionary elements as parameters.

Filter quantization

• Two questions:
  • How do we quantize?
  • Quantization $\rightarrow$ discrete values. How do we optimize?

• Example 2: binarize/ternarize
  • Weights $\rightarrow$ binary/ternary, + real-valued scale
  • Parameter updates happen in real space
