An Introduction to Optimization and Regularization Methods in Deep Learning

Yuan YAO
HKUST
Summary

- Last time: First order optimization methods
  - GD (BP), SGD, Nesterov, Adagrad, ADAM, RMSPROP, etc.
- This time
  - Second order methods
  - Regularization methods
  - Feifei Li, Stanford cs231n
Second Order Methods
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

- **Loss**
- **Epoch**

![Graph showing learning rate decay](graph.png)

More critical with SGD+Momentum, less common with Adam.
First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation

![Diagram showing the process of first-order optimization]
Second-Order Optimization

(1) Use gradient and Hessian to form quadratic approximation
(2) Step to the minima of the approximation
Newton Method

Second-Order Optimization

second-order Taylor expansion:

\[
J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0)
\]

Solving for the critical point we obtain the Newton parameter update:

\[
\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)
\]

Q: What is nice about this update?
Second-Order Optimization

second-order Taylor expansion:

\[
J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0)
\]

Solving for the critical point we obtain the Newton parameter update:

\[
\theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0)
\]

No hyperparameters!
No learning rate!

Q: What is nice about this update?
But, ...

Second-Order Optimization

second-order Taylor expansion:

\[ J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0) \]

Solving for the critical point we obtain the Newton parameter update:

\[ \theta^* = \theta_0 - H^{-1} \nabla_\theta J(\theta_0) \]

Hessian has \( O(N^2) \) elements
Inverting takes \( O(N^3) \)
\( N = \) (Tens or Hundreds of) Millions

Q2: Why is this bad for deep learning?
Second-Order Optimization

\[ \theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0) \]

- Quasi-Newton methods (BGFS most popular): instead of inverting the Hessian (\(O(n^3)\)), approximate inverse Hessian with rank 1 updates over time (\(O(n^2)\) each).

- L-BFGS (Limited memory BFGS): Does not form/store the full inverse Hessian.
L-BFGS

- **Usually works very well in full batch, deterministic mode**
  i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

In practice:

- **Adam** is a good default choice in most cases

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)
Regularizations
Regularization: Add term to loss

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W) \]

In common use:

- **L2 regularization**
  \[ R(W) = \sum_k \sum_l W_{k,l}^2 \] (Weight decay)

- **L1 regularization**
  \[ R(W) = \sum_k \sum_l |W_{k,l}| \]

- **Elastic net (L1 + L2)**
  \[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
Regularization: Dropout

In each forward pass, randomly set some neurons to zero.
Probability of dropping is a hyperparameter; 0.5 is common.

Regularization: Dropout

\( p = 0.5 \) # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
Regularization: Dropout
How can this possibly be a good idea?

Forces the network to have a redundant representation;
Prevents co-adaptation of features

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

score

X
Dropout as random perturbations of models

Regularization: Dropout
How can this possibly be a good idea?

Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...
Dropout: Test time

Dropout makes our output random!

\[ y = f_W(x, z) \]

Want to “average out” the randomness at test-time

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]

But this integral seems hard …
Dropout: Test time

Want to approximate the integral

\[ y = f(x) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz \]

Consider a single neuron.

At test time, we have:

\[ E[a] = w_1x + w_2y \]

During training, we have:

\[ E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) = \frac{1}{2}(w_1x + w_2y) \]

At test time, **multiply by dropout probability**
Dropout: Test time

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time
```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

Dropout Summary

- **Dropout in Forward Pass**: Drop units with probability \( p \) in the forward pass.
- **Scaling at Test Time**: Scale the activations by \( p \) at test time to compensate for the dropped units.
More common: “Inverted dropout”

```python
p = 0.5  # probability of keeping a unit active. higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p  # first dropout mask. Notice /p!
    H1 *= U1  # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p  # second dropout mask. Notice /p!
    H2 *= U2  # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)  # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

*test time is unchanged!*
Data normalization

Step 1: Preprocess the data
(Imulate $X_{[NxD]}$ is data matrix, each example in a row)

Remember: Consider what happens when the input to a neuron is always positive...
What can we say about the gradients on $w$?

Always all positive or all negative :-(
(this is also why you want zero-mean data!)

(Assume $X_{[NxD]}$ is data matrix, each example in a row)

$$f\left(\sum_i w_i x_i + b\right)$$
Data normalization

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

After normalization: less sensitive to small changes in weights; easier to optimize
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening
Batch Normalization

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

this is a vanilla differentiable function…
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

[Ioffe and Szegedy, 2015]
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

\[
\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

[Ioffe and Szegedy, 2015]
Batch Normalization

 Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = E[x^{(k)}]$$

to recover the identity mapping.

[Ioffe and Szegedy, 2015]
Batch Normalization

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

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

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

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

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \quad \text{// normalize}$$

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[Ioffe and Szegedy, 2015]
Batch Normalization

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

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

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

**Note:** at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

[ioffe and Szegedy, 2015]
Regularization: Data Augmentation

Load image and label

“cat”

CNN

Compute loss

This image by Nika is licensed under CC-BY 2.0
Regularization: Data Augmentation

- Load image and label
- "cat"
- Transform image
- CNN
- Compute loss
**Data Augmentation**

**Random crops and scales**

**Training**: sample random crops / scales

ResNet:
1. Pick random $L$ in range $[256, 480]$
2. Resize training image, short side $= L$
3. Sample random $224 \times 224$ patch

**Testing**: average a fixed set of crops

ResNet:
1. Resize image at 5 scales: $\{224, 256, 384, 480, 640\}$
2. For each size, use 10 $224 \times 224$ crops: 4 corners + center, + flips
Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in [Krizhevsky et al. 2012], ResNet, etc)
Data Augmentation

Get creative for your problem!

Random mix/combinations of:
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)
Regularization: A common pattern

**Training:** Add some kind of randomness

\[ y = f_W(x, z) \]

**Testing:** Average out randomness (sometimes approximate)

\[ y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz \]
Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling

Graham, “Fractional Max Pooling”, arXiv 2014
Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

**Examples**:
- Dropout
- Batch Normalization
- Data Augmentation
- DropConnect
- Fractional Max Pooling
- Stochastic Depth

Review: LeNet-5

[LeCun et al., 1998]

Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]
Popular Architectures

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

- ILSVRC’15 ResNet: 3.57 layers
- ILSVRC’14 GoogleNet: 6.7 layers
- ILSVRC’14 VGG: 7.3 layers
- ILSVRC’13: 11.7 layers
- ILSVRC’12 AlexNet: 16.4 layers
- ILSVRC’11: shallow
- ILSVRC’10: 28.2 layers

152 layers
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

First CNN-based winner
Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
- CONV1
- MAX POOL1
- NORM1
- CONV2
- MAX POOL2
- NORM2
- CONV3
- CONV4
- CONV5
- Max POOL3
- FC6
- FC7
- FC8

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Deeper Networks

Figure copyright Kaiming He, 2016. Reproduced with permission.
Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:
- ILSVRC’14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks
### Case Study: VGGNet

[Simonyan and Zisserman, 2014]

**Small filters, Deeper networks**

8 layers (AlexNet)  
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1  
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC’13  
(ZFNet)  
-> 7.3% top 5 error in ILSVRC’14

---

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Filters</th>
<th>Stride</th>
<th>Pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FC 4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Filters</th>
<th>Stride</th>
<th>Pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FC 4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Filters</th>
<th>Stride</th>
<th>Pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>64</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FC 4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Layer Type</th>
<th>Filters</th>
<th>Stride</th>
<th>Pad</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>128</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>256</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3x3 conv</td>
<td>512</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Pool</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FC 4096</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC 1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Softmax</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

AlexNet | VGG16 | VGG19
**Case Study: VGGNet**

*Simonyan and Zisserman, 2014*

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 \times (3^2C^2)$ vs. $7^2C^2$ for C channels per layer
INPUT: [224x224x3] memory: 224*224*3=150K  params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: 224*224*64=3.2M  params: (3*3*3)*64 = 1,728

CONV3-64: [224x224x64] memory: 224*224*64=3.2M  params: (3*3*64)*64 = 36,864

POOL2: [112x112x64] memory: 112*112*64=800K  params: 0

CONV3-128: [112x112x128] memory: 112*112*128=1.6M  params: (3*3*64)*128 = 73,728

CONV3-128: [112x112x128] memory: 112*112*128=1.6M  params: (3*3*128)*128 = 147,456

POOL2: [56x56x128] memory: 56*56*128=400K  params: 0

CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*128)*256 = 294,912

CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*256)*256 = 589,824

CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*256)*256 = 589,824

POOL2: [28x28x256] memory: 28*28*256=200K  params: 0

CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*256)*512 = 1,179,648

CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*512)*512 = 2,359,296

CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*512)*512 = 2,359,296

POOL2: [14x14x512] memory: 14*14*512=100K  params: 0

CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296

CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296

POOL2: [7x7x512] memory: 7*7*512=25K  params: 0

FC: [1x1x4096] memory: 4096  params: 7*7*512*4096 = 102,760,448

FC: [1x1x4096] memory: 4096  params: 4096*4096 = 16,777,216

FC: [1x1x1000] memory: 1000  params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

Deeper Networks

ILSVRC'15 ResNet
ILSVRC'14 GoogleNet
ILSVRC'14 VGG
ILSVRC'13
ILSVRC'12 AlexNet
ILSVRC'11
ILSVRC'10

3.57
6.7
7.3
11.7
16.4
25.8
28.2

152 layers

Figure copyright Kaiming He, 2016. Reproduced with permission.
Case Study: GoogLeNet
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC’14 classification winner
  (6.7% top 5 error)
Case Study: GoogLeNet

[Szegedy et al., 2014]

“Inception module”: design a good local network topology (network within a network) and then stack these modules on top of each other.
ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

“Revolution of Depth”

Figure copyright Kaiming He, 2016. Reproduced with permission.
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC’15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC’15 and COCO’15!
Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

56-layer model performs worse on both training and test error

-> The deeper model performs worse, but it’s not caused by overfitting!
Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.
Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

\[
H(x) = F(x) + x
\]

“Plain” layers

Use layers to fit residual
\[
F(x) = H(x) - x
\]

instead of \( H(x) \) directly
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:
- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)

$$F(x) + x$$

Residual block
Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or 152 layers for ImageNet
Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)
Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used
Case Study: ResNet

[He et al., 2015]

Experimental Results
- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "Ultra-deep" (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

ILSVRC 2015 classification winner (3.6% top 5 error) -- better than “human performance”! (Russakovsky 2014)
Improving ResNets...

Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time
Improving ResNets...
Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module
Reference


Reference


RefERENCE


Thank you!