CSE 291: Advances in Computer Vision

Manmohan Chandraker

Lecture 2: Background
Recap
Features have been key

SIFT [Lowe IJCV 04]

HOG [Dalal and Triggs CVPR 05]

SPM [Lazebnik et al. CVPR 06]

Textons

and many others:

SURF, MSER, LBP, GLOH, .....
Learning a Hierarchy of Feature Extractors

- Hierarchical and expressive feature representations
- Trained end-to-end, rather than hand-crafted for each task
- Remarkable in transferring knowledge across tasks
Significant recent impact on the field

Big labeled datasets
Deep learning
GPU technology

Error rates on ImageNet Visual Recognition Challenge, %

Sources: ImageNet; Stanford Vision Lab
Neuron

- Inputs are **feature values**
- Each feature has a **weight**
- Sum is the **activation**

\[
\text{activation}_w(x) = \sum_i w_i x_i = w \cdot x
\]

- If the activation is:
  - Positive, output +1
  - Negative, output -1

Slide credit: Pieter Abeel and Dan Klein
Two-layer neural network

\[ h_{\mathbf{w}}(\mathbf{x}) \]

\[ z \rightarrow \frac{1}{1 + e^{-z}} \]
Activation functions

**Sigmoid**

$\sigma(x) = \frac{1}{1 + e^{-x}}$

**tanh**  $\tanh(x)$

**ReLU**  $\max(0, x)$

**Leaky ReLU**

$\max(0.1x, x)$

**ELU**

$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$
From fully connected to convolutional networks

image

Fully connected layer

input layer  hidden layer 1  hidden layer 2  hidden layer 3

output layer
From fully connected to convolutional networks
From fully connected to convolutional networks

Slide: Lazebnik
From fully connected to convolutional networks

image

Convolutional layer

next layer
Learnable filters

- Several, handcrafted filters in computer vision
  - Canny, Sobel, Gaussian blur, smoothing, low-level segmentation, morphological filters, Gabor filters

- Are they optimal for recognition?
- Can we learn them from our data?
Number of weights

How many weights for this neuron?

$7 \cdot 7 \cdot 3 = 147$
Filters over the whole image

Assume the image is 30x30x3. 1 filter every pixel (stride = 1)
How many parameters in total?

- 24 filters along the x axis
- 24 filters along the y axis
- Depth of 5
- $7 \times 7 \times 3$ parameters per filter

423K parameters in total
Weight sharing

Insight: Images have similar features at various spatial locations!

- So, if we are anyways going to compute the same filters, why not share?
  - Sharing is caring

Assume the image is 30x30x3. 1 column of filters common across the image. How many parameters in total?

Depth of 5
\[ \times \ 7 \times 7 \times 3 \] parameters per filter

735 parameters in total
Pooling operations

- Aggregate multiple values into a single value
- Invariance to small transformations
  - Keep only most important information for next layer
- Reduces the size of the next layer
  - Fewer parameters, faster computations
- Observe larger receptive field in next layer
  - Hierarchically extract more abstract features
Convolutional Neural Networks
Architectural details of AlexNet

18.2% error in Imagenet

- Similar framework to LeCun 1998 but:
  - Bigger model (7 hidden layers, 650k units, 60M parameters)
  - More data ($10^6$ images instead of $10^3$ images)
  - GPU implementation (50 times speedup over CPU)
VGGNet architecture

- Much more accurate
  - AlexNet: 18.2% top-5 error
  - VGGNet: 6.8% top-5 error
- More than twice as many layers
- Filters are much smaller
- Harder and slower to train

### Table 2: Number of parameters (in millions).

<table>
<thead>
<tr>
<th>Network</th>
<th>A,A-LRN</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters</td>
<td>133</td>
<td>133</td>
<td>134</td>
<td>138</td>
<td>144</td>
</tr>
</tbody>
</table>
Deep residual learning

- Simple design
- Use only 3x3 conv (like VGG)
- No hidden FC
Key ideas for CNN architectures

• Convolutional layers
  – Same local functions evaluated everywhere
  – Much fewer parameters

• Pooling
  – Larger receptive field

• ReLU
  – Maintain a gradient signal over large portion of domain

• Limit parameters
  – Sequence of 3x3 filters instead of large filters
  – 1x1 convolutions to reduce feature dimensions

• Skip network
  – Easier optimization with greater depth
Optimization in CNNs
A 3-layer network for digit recognition

- Each grayscale image is of size 28x28.
- 60,000 training images and 10,000 test images
- 10 possible labels (0,1,2,3,4,5,6,7,8,9)

MNIST dataset

Example outputs:
6 -> [0 0 0 0 0 1 0 0 0]
Cost function

\[ C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \]

- The network tries to approximate the function \( y(x) \) and its output is \( a \).
- We use a quadratic cost function, or MSE, or “L2-loss”.

parameters to compute

\# of input samples

input -> \( x \)

vector output -> \( a \)
Gradient descent

\[ C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \]

- **parameters to compute**
- **# of input samples**

\[ \Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \]

Small changes in parameters to leads to small changes in output

\[ \nabla C \equiv \left( \frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T \]

Gradient vector!

\[ \Delta v = -\eta \nabla C \]

Change the parameter using learning rate (positive) and gradient vector!

\[ v \rightarrow v' = v - \eta \nabla C \]

Update rule!
Stochastic gradient descent

Cost function is a sum over all the training samples:

\[ C(w, b) = \sum_x C_x(w, b), \quad \text{where} \quad C_x(w, b) = \frac{1}{2} \| y(x) - a \|^2 \]

Gradient from entire training set:

\[ \nabla C = \frac{1}{n} \sum_x \nabla C_x \]

Update rules for each parameter:

\[ w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \]

\[ b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \]

Usually, \( n \) is very large.
Stochastic gradient descent

Gradient from entire training set:

\[ \nabla C = \frac{1}{n} \sum_x \nabla C_x \]

- For large training data, gradient computation takes a long time
  - Leads to "slow learning"

- Instead, consider a mini-batch with \( m \) samples
- If sample size is large enough, properties approximate the dataset

\[ \frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C. \]
Stochastic gradient descent

What if loss changes quickly in one direction and slowly in another? What does gradient descent do? Very slow progress along shallow dimension, jitter along steep direction.

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large.
What if the loss function has a local minima or saddle point?

Zero gradient, gradient descent gets stuck
Stochastic gradient descent

Our gradients come from minibatches so they can be noisy!

\[
L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)
\]

\[
\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)
\]
Stochastic gradient descent

Momentum update:

Velocity

actual step

Gradient

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

Build up velocity as a running mean of gradients.
Backpropagation

• In order to differentiate a function $z = f(g(x))$ w.r.t $x$, we can do the following:

Let $y = g(x)$, $z = f(y)$, \[ \frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx} \]

Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and $f$ maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $y = g(x)$ and $z = f(y)$, then

\[ \frac{\partial z}{\partial x_i} = \sum_k \frac{\partial z}{\partial y_k} \frac{\partial y_k}{\partial x_i} \]

This is all you need to know to get the gradients in a neural network!

Backpropagation: application of chain rule in certain order, taking advantage of forward propagation to efficiently compute gradients.
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
e.g. \( x = -2, y = 5, z = -4 \)

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want:
\[ \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \]
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ \text{e.g. } x = -2, y = 5, z = -4 \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \[ \frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z} \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ \begin{align*}
q &= x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \\
\end{align*} \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} \]
Backpropagation example

Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x} \]

Add gate: gradient distributor
Mul gate: gradient switcher
Convolutional layer is differentiable

- Activation function
  \[ a_{rc} = \sum_{i=-a}^{a} \sum_{j=-b}^{b} x_{r-i,c-j} \cdot \theta_{ij} \]

- Essentially a dot product, similar to linear layer
  \[ a_{rc} \sim x_{\text{region}}^T \cdot \theta \]

- Gradient w.r.t. the parameters
  \[ \frac{\partial a_{rc}}{\partial \theta_{ij}} = \sum_{r=0}^{N-2a} \sum_{c=0}^{N-2b} x_{r-i,c-j} \]
Max Pooling

- Run a sliding window of size $[h_f, w_f]$
- At each location keep the maximum value
- Activation function: $i_{\text{max}}, j_{\text{max}} = \arg \max_{i,j} x_{ij} \rightarrow a_{rc} = x[i_{\text{max}}, j_{\text{max}}]$
- Gradient w.r.t. input $\frac{\partial a_{rc}}{\partial x_{ij}} = \begin{cases} 1, & \text{if } i = i_{\text{max}}, j = j_{\text{max}} \\ 0, & \text{otherwise} \end{cases}$
- The preferred choice of pooling
Loss Functions and Regularizations
Slow learning with sigmoid neurons

- Learning is slow is same as saying that the “partial derivatives” are small.
- Consider the cost function and partial derivatives:

\[ C = \frac{(y-a)^2}{2}, \quad a = \sigma(z), \quad z = wx + b \]

\[ \frac{\partial C}{\partial w} = (a - y)\sigma'(z)x, \quad \frac{\partial C}{\partial b} = (a - y)\sigma'(z) \]

sigmoid function
Slow learning with sigmoid neurons

- Learning is slow is same as saying that the “partial derivatives” are small: $\frac{\partial C}{\partial w} = a\sigma'(z), \frac{\partial C}{\partial b} = a\sigma'(z)$

When the neuron’s output is close to 1, learning becomes slow.
Cross-Entropy Loss

- $C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$
- $n$ is the total number of items of training data
- $x$ is the input
- $y$ is the required output and $a$ is the output from the neuron
Cross-Entropy Loss

\[
\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \frac{\partial \sigma(z)}{\partial w_j} = -\frac{1}{n} \sum_x \left( \frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)} \right) \sigma'(z)x_j
\]

\[
= \frac{1}{n} \sum_x (\sigma(z) - y)x_j
\]

- Rate of learning depends on error in prediction!
- Prevents the learning slowdown from derivative of sigmoid.
Better activation functions

- Computes $f(x) = \max(0, x)$
- Does not saturate (in positive region)
- Computationally efficient
- Converges faster than sigmoid

ReLU
(Rectified Linear Unit)

- Same advantages as ReLU
- Stays alive when $x < 0$

Leaky ReLU
$f(x) = \max(0.01x, x)$
Over-fitting

- Instead of 60000 training images, we use only 1000 training images and check the performance on the test data.
More data prevents over-fitting

But not always feasible to have more data that is relevant.
Regularization reduces over-fitting
L2 regularization

L2 regularization:

\[ C = -\frac{1}{n} \sum_{x_j} [y_j \ln a^L_j + (1 - y_i) \ln (1 - a^L_j)] + \frac{\lambda}{2n} \sum_w w^2 \]

- The first term is just the usual expression for cross-entropy.
- Here \( \lambda \) is the regularization parameter and \( n \) is the size of our training set.

Partial derivatives:

\[ \frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \]

Update rule:

\[ W \rightarrow W - \eta \frac{\partial C_0}{\partial w} - \eta \frac{\lambda}{n} W \]
L1 regularization:

L1 regularization:

\[ C = C_0 + \frac{\lambda}{n} \sum_w |w| \]

\( C_0 \) is the cross-entropy term.

Partial derivatives:

\[
\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} \text{sgn}(w)
\]

Update rule:

\[
w \rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} \text{sgn}(w)
\]
L2 or L1 regularization

- In L1 case, the weights shrink by a constant amount towards 0.
- In L2 case, the weights shrink by an amount that is proportional to $w$.
- When the weight has a large magnitude $|w|$, then the L1 regularization shrinks less than the L2.
- When the weight has a small magnitude $|w|$, then the L1 regularization shrinks more than the L2.
- The net result is that the L1 regularization focuses on the weights of a few important connections and the rest are driven to zero.
Regularization reduces over-fitting
Dropout as a regularization

• Modify the network itself
  – Randomly delete half the hidden neurons in the network
  – Repeat several times to learn weights and biases
  – At runtime, twice as many neurons, so halve the weights outgoing from a neuron
Dropout as a regularization

• Modify the network itself
  – Randomly delete half the hidden neurons in the network
  – Repeat several times to learn weights and biases
  – At runtime, twice as many neurons, so halve the weights outgoing from a neuron

• Averaging or voting scheme to decide output
  – Same training data, but random initializations
  – Each network over-fits in a different way
  – Average output not sensitive to particular mode
Dropout as a regularization

- A useful insight from AlexNet paper
  - Reduces complex co-adaptations of neurons, since a neuron cannot rely on presence of others
  - Each neuron forced to learn independent features in conjunction with random other neurons
  - Dropout ensures the model can make robust predictions.
Data augmentation as regularization

Load image and label → “cat” → Transform image → CNN → Compute loss
Data augmentation as regularization

Horizontal flips
Data augmentation as regularization

Random crops and scales

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

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
Data augmentation as regularization

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

Can do a lot more: rotation, shear, non-rigid, motion blur, lens distortions, ....
Transfer Learning in CNNs
Transfer Learning

- Improvement of learning in a **new** task through the **transfer of knowledge** from a **related** task that has already been learned.

- Weight initialization for CNN

---

Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks  \[Oquab et al. CVPR 2014\]

Slide: Jiabin Huang
Transfer Learning

- Assume two datasets, $T$ and $S$
  - Dataset $S$ is
    - fully annotated, plenty of images
    - We can build a model $h_s$
  - Dataset $T$ is
    - Not as much annotated, or much fewer images
    - The annotations of $T$ do not need to overlap with $S$
- We can use the model $h_s$ to learn a better $h_T$
- This is called transfer learning

Imagenet: 1 million

My dataset: 1,000
CNNs are good at transfer learning

- Even if our dataset $T$ is not large, we can train a CNN for it
- Pre-train a network on the dataset $S$
- Then, there are two solutions
  - Fine-tuning
  - CNN as feature extractor
Fine-tune $h_T$ using $h_S$ as initialization

- Assume the parameters of $S$ are already a good start near our final local optimum
- Use them as the initial parameters for our new CNN for the target dataset
  \[ \theta_{T,l}^{(t=0)} = \theta_{S,l} \] for some layers $l = 1, 2, \ldots$
- This is a good solution when the dataset $T$ is relatively big
  - E.g. for Imagenet $S$ with approximately 1 million images
  - For a dataset $T$ with more than a few thousand images should be ok
- What layers to initialize and how?
Initializing $h_T$ with $h_S$

- Classifier layer to loss
  - The loss layer essentially is the “classifier”
  - Same labels $\rightarrow$ keep the weights from $h_S$
  - Different labels $\rightarrow$ delete the layer and start over
  - When too few data, fine-tune only this layer

Diagram:
- Classifier layer fc8
- Fully connected layer fc7
- Fully connected layer fc6
- Convolutional Layer 5
- Convolutional Layer 4
- Convolutional Layer 3
- Convolutional Layer 2
- Convolutional Layer 1
Initializing $h_T$ with $h_S$

- Classifier layer to loss
  - The loss layer essentially is the “classifier”
  - Same labels $\rightarrow$ keep the weights from $h_S$
  - Different labels $\rightarrow$ delete the layer and start over
  - When too few data, fine-tune only this layer

- Fully connected layers
  - Very important for fine-tuning
  - Sometimes you need to completely delete the last before the classification layer if datasets are very different
  - Capture more semantic, “specific” information
  - Always try first when fine-tuning
  - If you have more data, fine-tune also these layers
Initializing $h_T$ with $h_S$

- Upper convolutional layers (conv4, conv5)
  - Mid-level spatial features (face, wheel detectors ...)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tuning pays off
  - Fine-tune if dataset is big enough
Initializing $h_T$ with $h_S$

- **Upper convolutional layers (conv4, conv5)**
  - Mid-level spatial features (face, wheel detectors ...)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tuning pays off
  - Fine-tune if dataset is big enough

- **Lower convolutional layers (conv1, conv2)**
  - They capture low level information
  - This information does not change usually
  - Probably, no need to fine-tune but no harm trying
Strategy for fine-tuning

- For layers initialized from $h_S$ use a mild learning rate
  - Remember: your network is already close to a near optimum
  - If too aggressive, learning might diverge
  - A learning rate of 0.001 is a good starting choice (assuming 0.01 was the original learning rate)

- For completely new layers (e.g. loss) use aggressive learning rate
  - If too small, the training will converge very slowly
  - Remember: the rest of the network is near a solution, this layer is very far from one
  - A learning rate of 0.01 is a good starting choice

- If datasets are very similar, fine-tune only fully connected layers

- If datasets are different and you have enough data, fine-tune all layers
Use $h_S$ as a feature extractor for $h_T$

- Essentially similar to a case of solution I
  - but train only the loss layer

- Essentially use the network as a pretrained feature extractor

- This is a good solution if the dataset $T$ is small
  - Any fine-tuning of layer might cause overfitting

- Or when we don’t have the resources to train a deep net

- Or when we don’t care for the best possible accuracy
Transfer learning is a common choice
Training a Good CNN
Verifying that CNN is Trained Well

- Check gradients numerically by finite differences
- Visualize features (feature maps need to be uncorrelated) and have high variance.
- Visualize parameters
- Measure error on both training and validation set.

[M. Ranzato]
Verifying that CNN is Trained Well

- **Training diverges:**
  - Learning rate may be too large → decrease learning rate
  - BPROP is buggy → numerical gradient checking

- **Parameters collapse / loss is minimized but accuracy is low**
  - Check loss function:
    - Is it appropriate for the task you want to solve?
    - Does it have degenerate solutions? Check “pull-up” term.

- **Network is underperforming**
  - Compute flops and nr. params. → if too small, make net larger
  - Visualize hidden units/params → fix optimization

- **Network is too slow**
  - Compute flops and nr. params. → GPU, distrib. framework, make net smaller

[M. Ranzato]