AlexNet

Pre process

1. Resize to 256 x 256 (special instructions).
2. Subtracting mean activity over the training set from each pixel.

Architecture

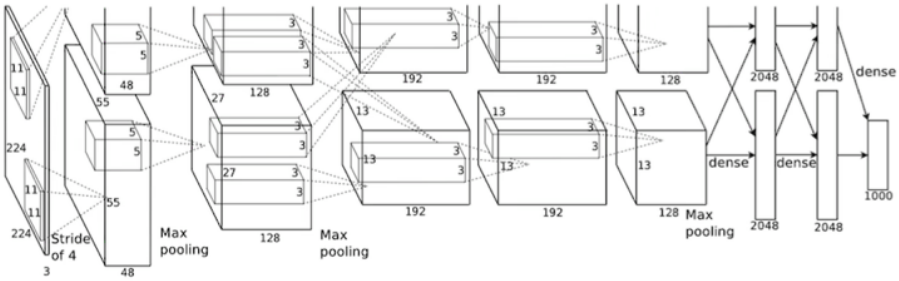
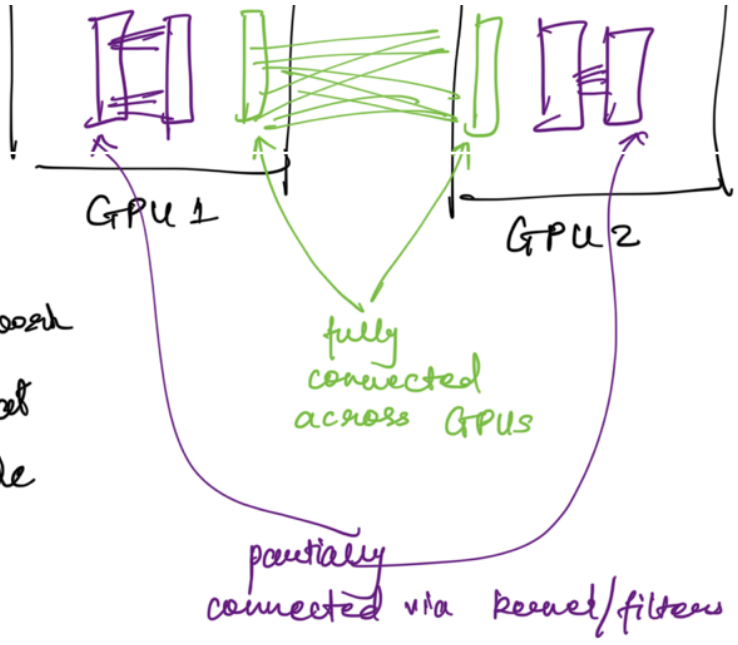


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

1. Using ReLU
2. Using some sort of GPU parallelizer → which is only for optimizing execution.

- However, for layer 4, each kernel only "sees" the layer 3 maps that are on the same GPU. In other words, the

same GPU. In other words, the network's third layer is fully connected across GPUs, but deeper layers are only partially connected—feature maps only interconnect within the same GPU [1] [2].



The "TRICK" is that they positioned network such that FC layers came at borders & conv. layers inside GPU.

= LRN (Local Response Normalization)

Breaking Down Your Formula Step by Step

Let's look at that scary formula:

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

What Each Symbol Means:

- $a^i_{x,y}$ = The original activation value of neuron i at position (x, y)
- $b^i_{x,y}$ = The new normalized value after LRN
- k = A small constant (like 2) to prevent division by zero
- α = Controls how strong the normalization effect is
- β = Controls the contrast (usually 0.75)
- n = How many neighbors to look at (usually 5)

The Process in Simple Steps:

1. **Pick a neuron** at position (x, y) in channel i
2. **Look at its neighbors:** Check n nearby channels (like 2 channels before and 2 channels after)
3. **Calculate neighbor strength:** Square all the neighbor values and add them up
4. **Create the denominator:** Add k plus α times the sum from step 3
5. **Divide:** Take the original value and divide by the denominator raised to power β

A Simple Example

Let's say you have 5 channels with activations at the same (x, y) position:

- Channel 0: 1.0
- Channel 1: 3.0 ← We want to normalize this one
- Channel 2: 2.0

So basically the "3.0" got normalized to "1.69".

- Channel 3: 1.5
- Channel 4: 0.5

Step 1: We're normalizing channel 1 (value = 3.0)
Step 2: Look at neighbors (let's say $n = 5$, so we look at all channels)
Step 3: Sum of squares =
$1.0^2 + 3.0^2 + 2.0^2 + 1.5^2 + 0.5^2 = 1 + 9 + 4 + 2.25 + 0.25 = 16.5$
Step 4: With $k = 2, \alpha = 0.0001, \beta = 0.75$:
Denominator = $(2 + 0.0001 \times 16.5)^{0.75} = (2.00165)^{0.75} 1.77$
Step 5: New value = $3.0/1.771.69 \simeq 1.69$

Overlapping Pooling

- traditionally we know it as non-overlapping → but they overlap it (similar to filter strides).

Architecture Details.

The first convolutional layer filters the $224 \times 224 \times 3$ input image with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring

[3]We cannot describe this network in detail due to space constraints, but it is specified precisely by the code and parameter files provided here: http://code.google.com/p/cuda-convnet/.

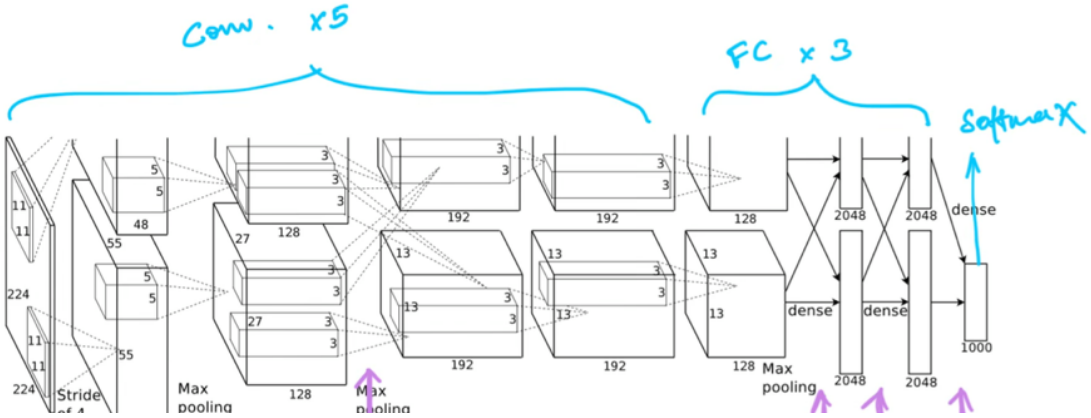


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

neurons in a kernel map). The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

Reducing Overfitting

⟹ The problem is 60 M parames & only 1.2 M data & based on info theory for a problem of 1000 classes we only get $\log_2(1000)$ data points

⇓

10 bits of data

∴ Wat data we have 1.2 M × 10 bits

= 12 M bits of data

∴ which is very less Compared to size of network.

Explained below ⇓

The "10 Bits of Constraint" Concept

This is where it gets mathematical, but here's the simple explanation:

- In **information theory**, if you have 1000 possible outcomes (classes), you need **$\log_2(1000)$ ≈ 10 bits** of information to specify which one is correct
- Each training example (image + its correct label) provides exactly this much "constraint" or "information" about what the network should learn
- So each training image only gives you **10 bits worth of guidance** for learning those 60 million parameters

The Core Problem: Not Enough Information

Here's the issue in simple terms:

Imagine this scenario: You're trying to solve a math problem with 60 million unknowns, but each clue you get only helps you figure out 10 bits worth of information. Even with 1.2 million training images (which AlexNet used), you're getting:

- **Total information**: 1.2 million × 10 bits = 12 million bits
- **Parameters to learn**: 60 million parameters
- **The gap**: You have far more things to learn than information to guide the learning

What This Leads To: Overfitting

Overfitting happens when your model becomes too specialized to the training data and fails to generalize to new, unseen data. With 60 million parameters and relatively little constraint per example, the network can:

- **Memorize** rather than **generalize**
- Learn irrelevant details specific to training images
- Perform great on training data but poorly on test data

↓ Solutions

1. Data Augmentation.

Step-by-Step Process

Step 1: Analyze All Training Images

- Collect **all RGB pixel values** from the entire ImageNet training set
- This gives you millions of R, G, B triplets

Step 2: Perform PCA (Principal Component Analysis)

- Find the **3 main directions** of color variation across all images
- These become your **eigenvectors** (p_1, p_2, p_3)
- Calculate how much variation occurs in each direction → **eigenvalues** (λ_1, λ_2, λ_3)

Understanding the Formula: $p_1, p_2, p_3 \alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3^T$

What each part means:

- **p_1, p_2, p_3**: The 3 eigenvectors (directions of color change)
- **λ_1, λ_2, λ_3**: The eigenvalues (how much change happens in each direction)
- **α_1, α_2, α_3**: Random numbers from a Gaussian distribution (mean=0, std=0.1)

Step 3: Apply to Each Training Image

For every pixel R, G, B in a training image:

1. **Generate random multipliers**: Draw α_1, α_2, α_3 from Gaussian(0, 0.1)
2. **Scale by eigenvalues**: Multiply each α_i by its corresponding λ_i
3. **Combine with eigenvectors**: Use the eigenvectors to convert back to RGB space
4. **Add to original pixel**: New pixel = Original + PCA adjustment

Transformations

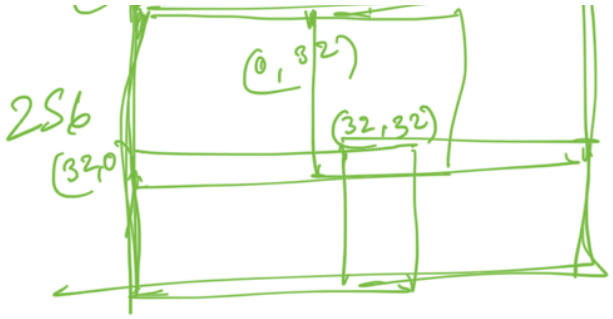
Train : Resize 256 x 256 | Crop 224 x 224

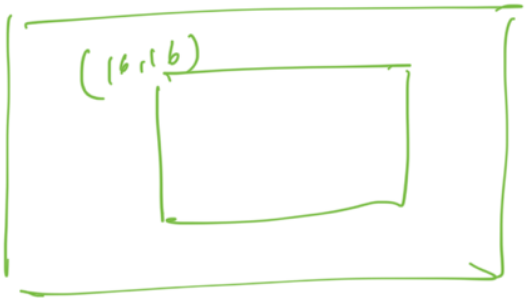
& Mirror images

Testing 224 x

224 Crops at 5 locations :

(0,0) 256





Second transformation → PCA

→ PCA → makes the model ROBUST to "lighting changes"

The Core Idea: Instead of randomly changing colors, PCA finds the **most common ways** that colors vary naturally in your dataset, then applies these "natural" color variations to create augmented training images.

we will come later

Data loaders

⟹ 2 main types of objects:

1. Dataset objects
2. Dataloader objects

} train & val

To verify we use a "Iterator" over a data_loader object.

LRN

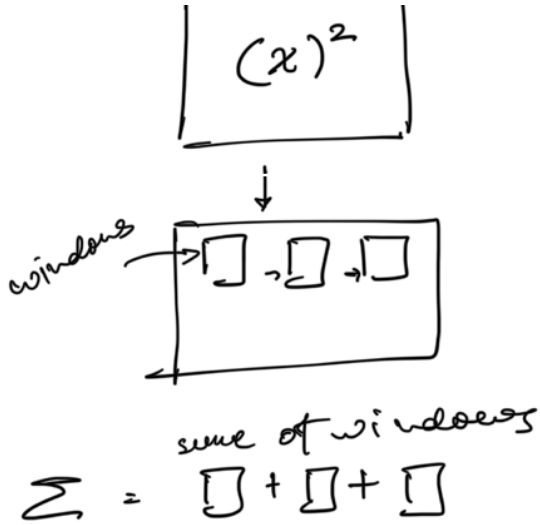
$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta}$$

- $a^i_{x,y}$ = Original activation at channel i, position (x, y)
- $b^i_{x,y}$ = Normalized activation after LRN
- k = Constant to prevent division by zero (typically 2.0)
- α = Controls normalization strength (typically 1e-4)
- β = Controls contrast (typically 0.75)
- n = Number of nearby channels to consider (typically 5)

input → x which is the input tensor

pad = 3

$$ans = x /$$



$$answer = x \Big/ (k + \alpha * \Sigma)^{\beta}$$

LRN Explanation

Understanding Data

```
python
batch_size, channels, height, width = x.shape
```

What this means: We're extracting the dimensions of our tensor.

Visual Example:

```
Let's say x.shape = [2, 5, 4, 4]
This means:
- batch_size = 2    (we're processing 2 images at once)
- channels = 5      (each image has 5 feature maps)
- height = 4        (each feature map is 4 pixels tall)
- width = 4         (each feature map is 4 pixels wide)
```

Real-world analogy: Think of this as having 2 photo albums, each album has 5 pages, and each page has a 4×4 grid of numbers.

Step 1: Square the Input

squaring value to EMPHASIZE the loudness of the activa^n → it also makes it +ve. (if -ve)

Step 2 : Padding

Step 2.1 : Calculate how much padding we need

```python
padding = self.size // 2
```

What this does: If size=5, then padding=2. This means we'll look at 2 channels before + current channel + 2 channels after = 5 total channels.

Why we need padding: Imagine you're in the middle of a row of people. You can look 2 people left and 2 people right. But what if you're at the edge? You need "imaginary" people (padding) so everyone gets fair treatment.

Step 2.2 : Add padding

Step 4: Adding Padding (The Tricky Part!)

```python
x_squared_padded = F.pad(x_squared, (0, 0, 0, 0, padding, padding))
```

What `(0, 0, 0, 0, padding, padding)` means:

- For a 4D tensor [batch, channels, height, width], padding works backwards:
- Last 2 numbers: pad width dimension (0, 0 = no padding)

- Next 2 numbers: pad height dimension (0, 0 = no padding)
- First 2 numbers: pad channel dimension (padding, padding)

Visual Example with padding=2:

```
Original tensor (5 channels):
[Ch0] [Ch1] [Ch2] [Ch3] [Ch4]

After padding (9 channels total):
[0] [0] [Ch0] [Ch1] [Ch2] [Ch3] [Ch4] [0] [0]
 ↑   ↑                                 ↑   ↑
pad pad                               pad pad
```

basically edge case handling

First Principle: We add "dummy" channels filled with zeros so that edge channels (Ch0 and Ch4) can also have neighbors to look at.

= can also refer this syntax to understand what is role of padding: width/height/channel with positions in an array.

```
# Padding operations
F.pad(tensor, pad_width, mode, value)
# pad_width = (left, right, top, bottom, front, back) for each dimension
# For [B,C,H,W], padding (0,0,0,0,2,2) pads only the channel dimension
```

Step 3: Sliding Window - for calculating sums in a window → And eventually finding value of "Σ" in the eq".

Step 3.1. "Unfold" operation finds the sliding window conditions automatically.

↓

Just the state → basically, what channels would be present in the window.

It finds out these lists

Step 5: Creating Sliding Windows (The Magic!)

```python
windows = x_squared_padded.unfold(1, self.size, 1)
```

What `unfold(1, self.size, 1)` does:

- `1` = operate on dimension 1 (channels)
- `self.size` = window size (how many channels to group together)
- `1` = step size (move 1 channel at a time)

Visual Example (size=3 for simplicity):

```
Padded channels: [0] [Ch0] [Ch1] [Ch2] [Ch3] [0]

Sliding windows:
Window for Ch0: [0, Ch0, Ch1]    ← channels that affect Ch0
Window for Ch1: [Ch0, Ch1, Ch2]  ← channels that affect Ch1
Window for Ch2: [Ch1, Ch2, Ch3]  ← channels that affect Ch2
Window for Ch3: [Ch2, Ch3, 0]    ← channels that affect Ch3
```

Detailed Shape Transformation:

```
Before unfold: x_squared_padded.shape = [batch, channels+padding*2, height, width]
                                      = [2, 5+2*2, 4, 4] = [2, 9, 4, 4]
```

```
After unfold:  windows.shape = [batch, channels, height, width, size]
                             = [2, 5, 4, 4, 5]
```

First Principle: Instead of manually creating loops to look at neighbors, `unfold` efficiently creates all the neighbor groups at once. It's like having a sliding magnifying glass that captures each channel and its neighbors.

Step 3-2 : For each window cond^n → Perform the sum.

Step 6: Sum Across the Window

```python
sum_of_squares = windows.sum(dim=-1)
```

What this does: For each channel at each position, add up the squared values of all its neighbors.

Visual Example:

```
For Channel 1 at position (0,0):
Window contains: [Ch0_squared, Ch1_squared, Ch2_squared, Ch3_squared, Ch4_squared]
Values might be: [4, 9, 1, 16, 0]
Sum = 4 + 9 + 1 + 16 + 0 = 30

This happens for every channel at every (height, width) position!
```

Shape after summing:

```
Before sum: [batch, channels, height, width, size] = [2, 5, 4, 4, 5]
After sum:  [batch, channels, height, width]      = [2, 5, 4, 4]
```

∴ based on eq^n → we have "Σ"

∴ finally we only need to execute

```
# Step 4: Apply the LRN formula
# denominator = (k + α * sum_of_squares)^β
denominator = torch.pow(self.k + self.alpha * sum_of_squares, self.beta)

# Avoid division by zero
denominator = torch.clamp(denominator, min=1e-8)

# Final normalization: b = a / denominator
normalized = x / denominator

return normalized
```

$b_{x,y}$

Architecture

Feature layers - Conv layers = 5

Classifier layers

- 2 FC layers
- Custom layer

Initializing weights

① for feature_layers

- normal-dist (mean = 0, std = 0.01)
- bias = 0

② for classifn_layers

- normal-dist (mean = 0, std = 0.01)
- bias = 1

Forward Prop.

1. send x through feature layers
   - after which we get a changed shape

```
# Before feature extraction:
# x.shape = [batch_size, 3, 224, 224]
#           [how many photos, RGB, height, width]
#           [4, 3, 224, 224] = 4 color photos, each 224x224 pixels

# After feature extraction:
# x.shape = [batch_size, 256, 6, 6]
#           [how many photos, observations, small regions, small regions]
#           [4, 256, 6, 6] = 4 analysis reports, each with 256 observations about 6x6 grid
```

2. The upcoming are the FC layers

which can't process grids of data

↓

∴ we FLATTEN it to turn the into list.

∴ from 4D → 2D

```
# TODO: Flatten the tensor for fully connected layers
# Step 2: flatten bcoz we need to convert from 4d tensor to 2d tensore

# Think: Convert from "image with features" to "list of features"


x.view(x.size(0), -1) # Keep batch size, flatten everything else

# continuation from example: turning the report into a list

# New shape: [batch_size, 256*6*6] = [batch_size, 9216]

#we have done this bcoz FC layers need a list of numbers and not grids
```

3. And then we finally pass it through classificar layers to get final outputs as the list of probs across no. of classes.

[0.20 , 0.0 , 0.1 , ... — .. 0.5]

The class with max prob. is the final result of our classification task.

Training Intra Setup

⟹ Setting up training components

1. loss function → cross entropy loss

"criterion" ← in code.

2. optimizer → Stochastic Gradient Descent

in code → "optimizer".

3. Learning Rate Scheduler.

in code → "scheduler".

⟹ Training one epoch

1. Initialize few things.
   - model.train() ← setting model to train
   - running_loss, correct, total.
   - shift "model, data, labels" to device.

2. Zero Gradients

3. Forward pass → just one line

4. Compute Loss

5. Backprop → compute gradient.

6. Optimizer → updates weights

7. Rest
   - stats for accuracy // loss // total batches done

⟹ Validate model

Similar code — just 2 main things:

1. Never Seen data

2. No backprop / learning
   → just pure monitoring

Driven

Done

Things to add

1. extract_10_test_crops.

2. Phase 8 with testing

3. Add PCA ⇒

Q. When we save a model → do we save it on "device"

↓

if not then why does

"load_trained_model" — have an arg. device.

Adding PCA on train data.

1. Denormalizing images

→ LATER : Q : was there a need of de normalizing — bcoz we are directly loading fresh data for pca.

```python
mean = torch.tensor([0.485, 0.456, 0.406])
std = torch.tensor([0.229, 0.224, 0.225])

for t, m, s in zip(images, mean, std):
    t.mul_(s).add_(m)
```

Why this is needed: When images are loaded for training, PyTorch "normalizes" them (changes their values to help training). But for PCA, we need the **original RGB color values.**

Analogy: It's like removing a color filter from photos to see their true colors.

The math:

- Original normalization: `normalized = (original - mean) / std`
- Undoing it: `original = normalized * std + mean`

2- Combining into all_pixels.

3. Concat into all_pixels

4 Math:

4.1 Centering

4.2 Covariance matrix

4.3 eigen-decomp.

4.4 Sorting

```python
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]
```

⟹ Part 2: PCA colour augmentation

This function actually EXERCISES PCA onto images.

⟹ Part 3: PCA driver → for making loaders lightweight

Done