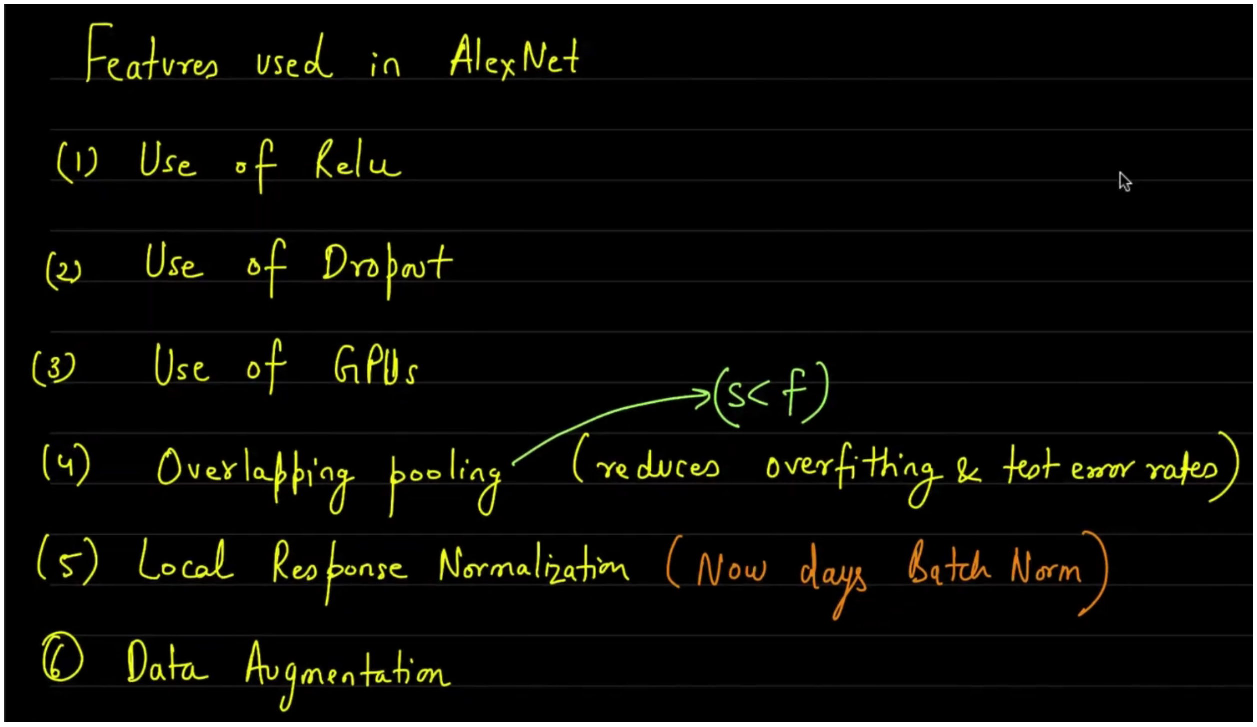
AlexNet



Features used in AlexNet

(1) Use of Relu

(2) Use of Dropout

(3) Use of GPUs

(4) Overlapping pooling (reduces overfitting & test error rates) $\rightarrow (s<f)$

(5) Local Response Normalization (Now days Batch Norm)

(6) Data Augmentation

Source: Ahlad Kumar - **Lecture: CNN Architectures (AlexNet, VGGNet, Inception ResNet)**

First Pass:

- Results
  - Top-5 error rate of 37.5%
  - Top-1 error rate of 17%
- Architecture Information
  - 60 million parameters
  - 5 Convolutional layers and 3 Fully Connected layers
- Discoveries for better training
  - They found dropout regularization as a powerful tool to reduce overfitting
  - Learning is always faster if you use ReLU activation instead of tanh in Conv layer because of the absence of saturation

    - The model depth is crucial as error rate increases significantly on removing any layer

Note: CNNs are a lot better than Fully Connected Networks for image tasks because they use way less parameters (weights and biases) than FC Networks and perform only slightly worse. This helps in decreasing the training time, avoiding overfitting and requiring less computational resources.

Second Pass:

The paper came out on September 30, 2012.

Large datasets were not very common back then. ImageNet and a few other competitions had recently started being used for benchmarking Machine Learning and Deep Learning models.

The paper focuses on the lack of complexity in the then deep learning models for such large datasets (ImageNet has over 15 million high-resolution images belonging to 22,000 categories). The paper also focuses on how the then used GPUs were just enough for models such as AlexNet with the optimizations they used such as parallelization with 2 GPUs.

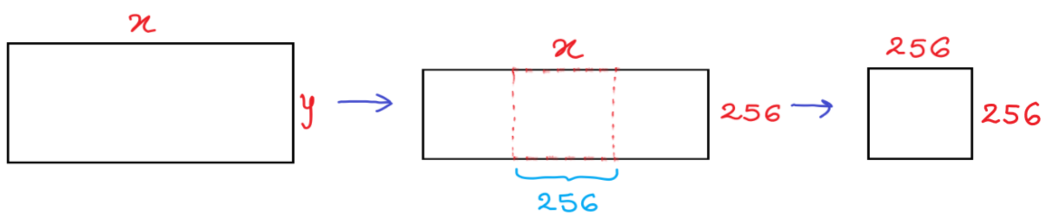
Overfitting was still a problem with a model like AlexNet even with 15 million high-resolution images.

About the dataset:

- Top-1 and Top-5 error rates are mandatory to report for the ImageNet dataset
- Only a subset of the ImageNet dataset is used in competitions/benchmarking containing 1000 classes and roughly 1000 training images per class as this in itself is plenty (1.2 million in total).

Data Preprocessing:

- Down-Sampling images to 227 x 227
- For rectangular images, the shorter side was made of length 256, and then they cropped out the central 256×256 patch from the resulting image



- Normalized/Centered the images by subtracting the mean of all the pixels

Third Pass:

Stress on overfitting and measures to combat it

The authors have tried to stress a lot on the problem of overfitting the dataset in the paper and have also tried various ways to prevent the same in AlexNet. However, a lot of the measures taken in the paper actually solve a different purpose and is not just overfitting.

Why are pooling layers used?

Using just Convolutional layers in ConvNets is a bad idea because the feature maps (kernels) focus on a single area at a time in the input image. This means the outputs of the kernels are very sensitive to the location where they are used. If the input image is even slightly shifted through any process (rotation, translation, shrinking, expanding, etc) then the kernels will give out different activations (outputs) and thus the result will vary greatly.

Thus, to avoid/reduce translational variance, a common approach from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

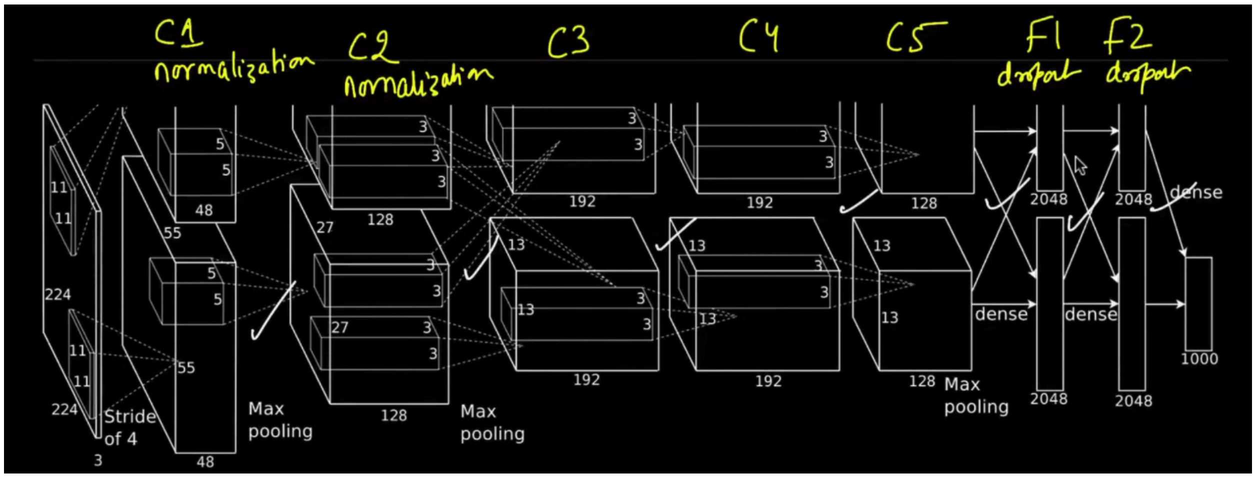
Thus, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.

Using Overlapping Pooling:

The authors found overlapping pooling to be more useful than non-overlapping pooling. This means a max-pooling layer of filter size (3 x 3) and stride = 2 was more effective than a max-pooling layer of filter size (2 x 2) and stride = 2.

Is overlapping pooling better in dealing with over-fitting in general?

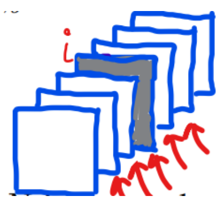
Architecture Discussion:



Source: Ahlad Kumar - **Lecture: CNN Architectures (AlexNet, VGGNet, Inception ResNet)**

Local Response Normalization:

The concept of Local Response Normalization is to use the activations of the neighboring kernels/filters and average them out to get the new activations for the current filter.



So, for the i^{th} filter, we will go from i - n/2 to i + n/2 filters, take the average of their activations to update the current layer.

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

This concept is not much used nowadays and has been replaced by ***Batch Normalization.*** We can also use *Layer Normalization* where we take the average of all the kernels/filters in the layer and not just the neighbors.

Dropout Regularization:

Using dropout regularization has been a trend for long. Dropout effectively reduces the architecture size while training and changes the architecture for each epoch.

A dropout layer with probability = 0.5 means that we will be dropping each neuron/unit in the layer with a probability of 0.5 and thus, not include it in the forward as well as the backward pass. This allows all the neurons to be less dependent on a single feature and learn more robust features. This will automatically reduce weights and thus have a similar effect as weight decay (L2).

This helps **reducing overfitting.**

However, a high value of dropout rate will also lead to slower convergence and increased parameters for learning in the model. We also cannot rule out the possibility of the model missing out on important trends/features altogether because of a high dropout rate.

Cross Validation will help in determining the best value of dropout for the task at hand.

Since dropout removes some of the units from a layer, a network with dropout will weigh the remaining units more heavily during each training run to compensate for the missing inputs. However, at test time, it is not feasible to use the weights of the trained model in their exaggerated states and so each weight is scaled down by multiplying by the hyperparameter *p.*

Inverted Dropout -

Define a keep_prob which is the probability of keeping a unit in any layer. So you will be dropping out units in a layer with a probability of (1 - keep_prob).

```
# Suppose keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
# This will cause 20% of the units to get dropped out/shut off.
# So we have to scale the output accordingly
a3 /= keep_prob
```

```
# We do this because z4 = w4 * a3 + b4. If a3 reduces, then the activations in the later
# layers will also get affected and thus we scale the dropped out layer
```

There is no need to perform any scaling at test time because you already scaled the outputs during training time which compensated for the dropout.

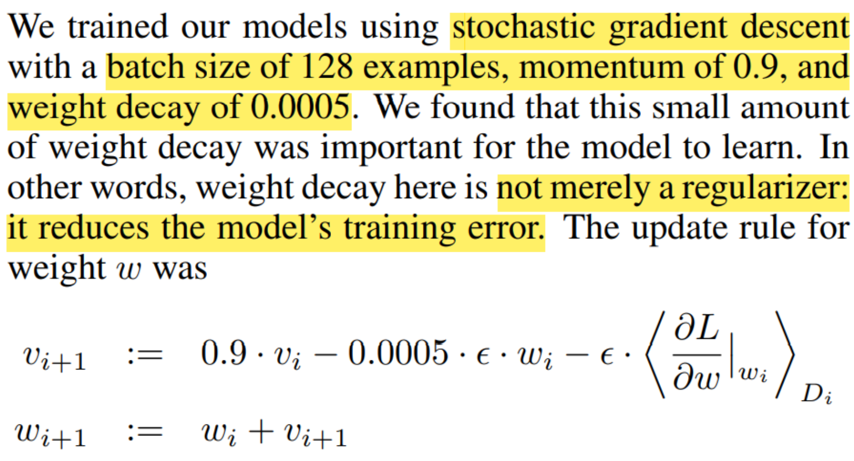
A better idea should be to use different keep_prob values for different layers. A layer which is more prone to overfitting (maybe it has a lot of weights) can be given a lower keep_prob value like 0.5 and layers that are less prone to overfitting can be given a higher keep_prob value, maybe even 1.0 if you don't worry about overfitting at all in some layers.

A big disadvantage of dropout is that since you don't have a well-defined architecture during the training time, it is harder to debug the training process with the help of a loss function. The loss function would simply be not well-defined. The way to get around this would be to pause the training, remove dropout from all the layers, train a little on the training data and see if the cost function is decreasing or not. If it is, then again put the dropout as it is and continue.

Data Augmentation:

Linear translations like cropping and flipping have been done to images for data augmentation.

Stochastic Gradient Descent with Momentum:



We trained our models using stochastic gradient descent with a batch size of 128 examples, momentum of 0.9, and weight decay of 0.0005. We found that this small amount of weight decay was important for the model to learn. In other words, weight decay here is not merely a regularizer: it reduces the model's training error. The update rule for weight w was

$$\begin{aligned} v_{i+1} &:= 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i} \\ w_{i+1} &:= w_i + v_{i+1} \end{aligned}$$

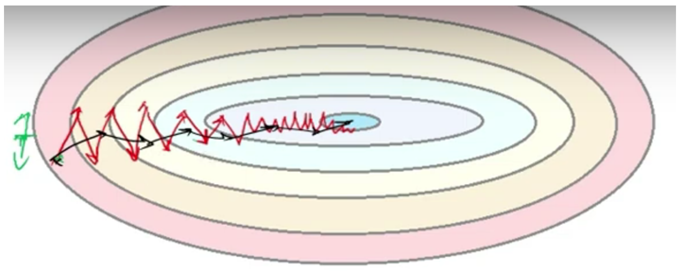
Momentum is a way of decreasing the jaggedness of the training process especially for stochastic and mini-batch gradient descent. It uses ***Exponentially Moving Weighted Averages*** for calculating the gradient.

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

It thus uses the previous gradient update values as well as the current gradient calculated by the algorithm for making an update to the weights.

A value of β = 0.9 means that the exponentially weighted average will only take into account roughly the last 1/(1-β) iterations. So a β = 0.9 will account for the last 10 iterations.

Taking an exponentially weighted average like this helps in cancelling out the 'vertical' deviations as they get averaged and helps in increasing the 'horizontal' deviations which are necessary for advancing towards the global optimum.



Source: Coding Lane **- Momentum Optimizer in Deep Learning | Explained in Detail**

A higher value of β puts more emphasis on the previous gradient values and less on the current gradient whereas a lower value of β puts less emphasis on the previous gradient values and more on the current gradient. A standard value of β chosen in Deep Learning is 0.9.

Gradient Descent with momentum is performed with the help of the following pseudocode.

$$v_{dw} = \beta v_{dw} + (1-\beta)dW$$
$$v_{db} = \beta v_{db} + (1-\beta)db$$
$$W = W - \alpha v_{dw}, \quad b = b - \alpha v_{db}$$

One more thing that can be implemented is ***bias correction*** for momentum. When calculating the initial gradients using momentum, the previous weighted averages would be close to zero since we start by initializing the V_{dw} = 0. This combined with a larger value of β like 0.9 will cause a diminished weight update during the early stages. This can be countered by scaling the weight update by dividing the exponentially moving average for the iteration t by $1 - \beta^t$.

Bias Correction:

$$V_{dw} = \frac{V_{dw}}{1 - \beta^t}$$

This will cause the average to increase during the earlier iterations. Its effect will wear off after some iterations, thus solving the purpose. For β = 0.9, the bias correction will be effective only for the first 10-15 iterations. This can be ignored because this will only effect the first few iterations and then our weight updations will become effective.