Training

Split

- Validation Data
- Train Data
- Test Data
    - Especially if hyperparams are tuned

Hyperparameter

- We define a hyper-parameter for a learning algorithm A as a variable to be set prior to the actual application of A to the data, one that is not directly selected by the learning algorithm itself.

- The hyper-parameters can be fixed by hand or tuned by an algorithm, but their value has to be selected.

- Ofcourse, one can hide these hyper-parameters by wrapping another learning algorithm, say B, around A, toselects A's hyper-parameters (e.g. to minimize validation set error).

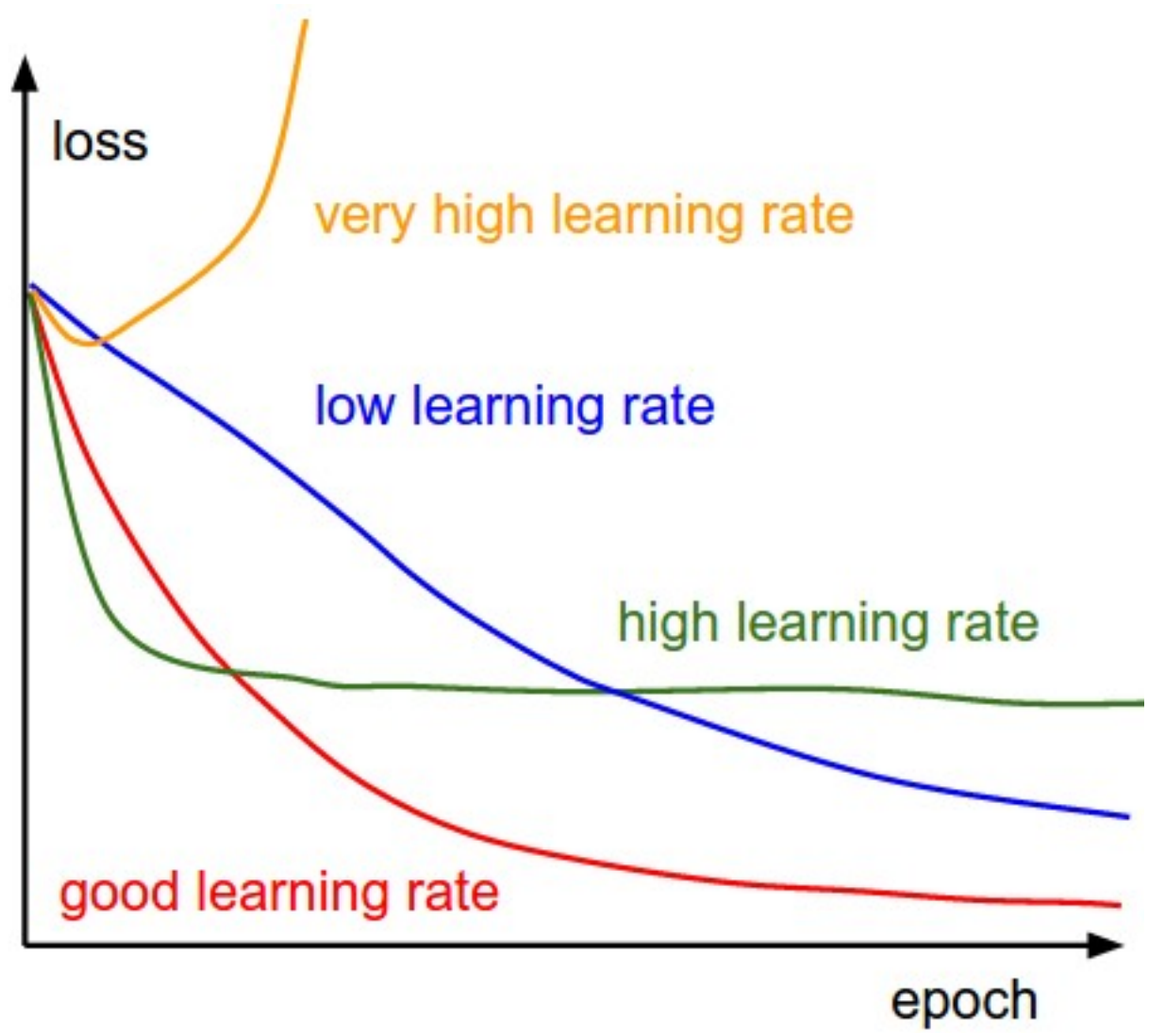
Learning rate

- A If the learning rate is too large, the average loss will increase.The optimal learning rate is usually close to (by a factor of 2) the largest learning rate that does not cause divergence of the training criterion, an observation that can guide heuristics for setting the learning rate (Bengio, 2011), e.g., start with a large learning rate and if the training criterion diverges, try again with 3 times smaller learning rate.

- This is often the single most important hyper-parameter and one should always make sure thatit has been tuned (up to approximately a factor of 2). Typical values for a neural network with standardized inputs (or inputs mapped tothe (0,1) interval) are less than 1 and greater than 10^-6

- The choice of strategy for decreasing or adapting the learning rate schedule. It could decrease with every iteration in an exponential manner or step manner

$$\epsilon_t = \frac{\epsilon_0 \tau}{max(t, \tau)}$$



Batching

- When batch size: B increases we can get more multiply-add operations per second by taking advantage of parallelism or efficient matrix-matrix multiplications (instead of separate matrix-vector multiplications), often gaining a factor of 2 in practice in overall training time. On the other hand, as B increases, the number of updates per computation done decreases, which slows down convergence (in terms of error vs number of multiply-add operations performed) because less updates can be done in the same computing time. Combining these two opposing effects yields a typical U-curve with a sweet spot at an intermediate value of B

- doing more updates more frequently helps to explore more and faster, especially with large learning rates. In addition, smaller values of B may benefit from more exploration in parameter space and a form of regularization both due to the“noise” injected in the gradient estimator, which may explain the better test results sometimes observed with smaller B

- The great advantage of stochastic gra-dient descent and other online or minibatch update methods is that their convergence does not depend on the size of the training set, only on the number of updates and the richness of the training distribution.As for any stochastic gradient descent method (including the mini-batch case), it is important for efficiency of the estimator that each example or mini-batch be sampled approximately independently.

- In this context, it is safer ifthe examples or mini-batches are first put in a ran-dom order (to make sure this is the case, it couldbe useful to first shuffle the examples).

-

Number of Training iterations

- This hyper-parameteris particular in that it can be optimized almost for free using the principle of early stopping: by keeping track of the out-of-sample error (as for example estimated on a validation set) as train-ing progresses (every N updates), one can decide how long to train for any given setting of all the other hyper-parameters.

- It might be use-ful to turn early-stopping off when analyzing the effect of individual hyper-parameters.

- Practically at candidate point of early stopping one would train P (patience) samples further and observe validation loss. If decrease then new candidate early stopping point and P also increased.

Momentum

- As you go down a hill, not only the current step, but past steps (through momentum) can guide you.

- Keep some smooth average $\hat{g} = \beta\hat{g} + (1 - \beta)g$

Hyper parameters of model

- Number of hidden units: Because of early stopping and possibly other regularizers (e.g., weight decay, discussed below), it is mostly important to choosen large enough. Larger than optimal values typically donot hurt generalization performance much, but of course they require proportionally more computation

- Weight decay: Away to reduce overfitting is to add aregulariza-tion term to the training criterion, which lim-its the capacity of the learner. The parameters of machine learning models can be regularized by pushing them towards a prior value, which is typically 0.

- Neuron nonlinearity: Sigmoid, Relu, tanh: For output (or reconstruction) units,hard neuron non-linearities like the rectifier donot make sense because when the unit is satu-rated (e.g.a <0 for the rectifier) and associ-ated with a loss, no gradient is propagated in-side the network, i.e., there is no chance to cor-rect the error. For output units a good trick is to obtain theoutput non-linearity and the loss by consideringthe associated negative log-likelihood and choos-ing an appropriate (conditional) output proba-bility model, usually in the exponential family.

- Weight initialisation: Biases can generally be initialized to zero but weights need to be initialized carefully to break the symmetry between hidden units of the same layer

    - recommend scaling std of random distribution by the inverse of the square root of the fan-in,

Fine-tuning (transfer learning)

- F**eature extractor**: Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier.

- **Finetuning**: The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it's possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network.

- **Pretrained models**. Since modern ConvNets take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final ConvNet checkpoints for the benefit of others who can use the networks for fine-tuning. For example, the Caffe library has a Model Zoo where people share their network weights

When to fine-tune

1. *New dataset is small and similar to original dataset*. Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.
2. *New dataset is large and similar to the original dataset*. Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.
3. *New dataset is small but very different from the original dataset*. Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier form the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.
4. *New dataset is large and very different from the original dataset*. Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pretrained model. In this case, we would have enough data and confidence to fine-tune through the entire network

- It's common to use a smaller learning rate for ConvNet weights that are being fine-tuned, in comparison to the (randomly-initialized) weights for the new linear classifier that computes the class scores of your new dataset. This is because we expect that the ConvNet weights are relatively good, so we don't wish to distort them too quickly and too much (especially while the new Linear Classifier above them is being trained from random initialization).

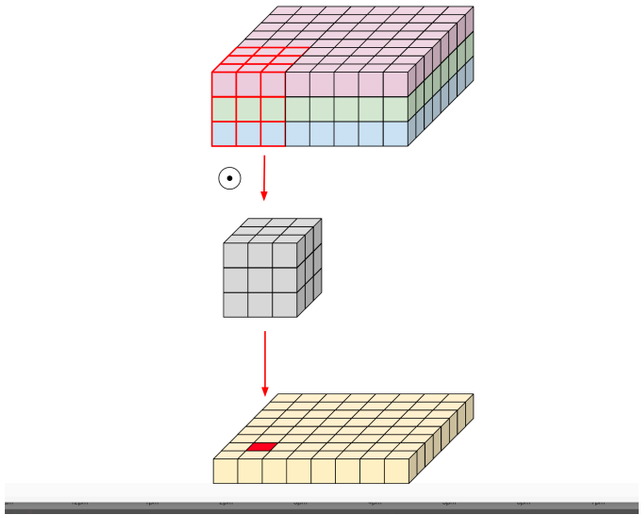
Fc to convolution

- It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical.

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing)

- Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with $K=4096$ $K = 4096$ that is looking at some input volume of size $7×7×512$ $7 \times 7 \times 512$ can be equivalently expressed as a CONV layer with $F=7,P=0,S=1,K=4096$ $F = 7 , P = 0 , S = 1 , K = 4096$. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be $1×1×4096$ $1 \times 1 \times 4096$ since only a single depth column "fits" across the input volume, giving identical result as the initial FC layer.

Separable 3D Convolutioon



- Top figure — conventional ‘3D’ convolution where a 3D filter is moved over the 3D tensor to give a 2D tensor

- For multiple filters these output 2D tensors will be stacked

- In the regular 2D convolution performed over multiple input channels, the filter is as deep as the input and lets us freely mix channels to generate each element in the output.

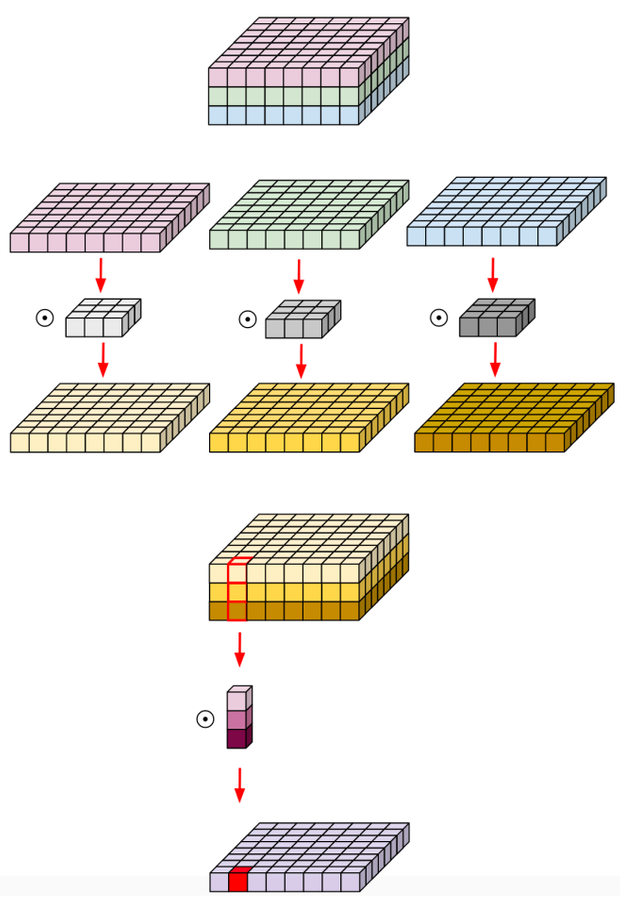
- Depthwise convolutions don't do that - each channel is kept separate - hence the name *depthwise*.

- depthwise separable convolution pictured here : After completing the depthwise convolution, and additional step is performed: a 1x1 convolution across channels. This is exactly the same operation as the "convolution in 3 dimensions discussed earlier" - just with a 1x1 spatial filter.

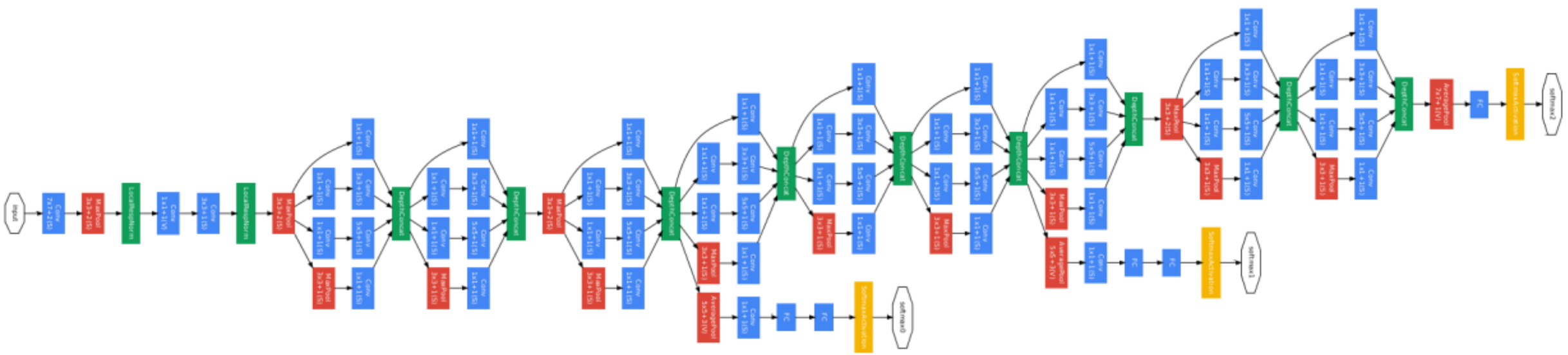
- Depthwise separable convolutions have become popular in DNN models recently, for two reasons

    - They have fewer parameters than "regular" convolutional layers, and thus are less prone to overfitting. F*F*inC*outC >> F*F*inC+ inC*outC

    - With fewer parameters, they also require less operations to compute, and thus are cheaper and faster.
      F*F*inC*S*S*outC>>F*F*inC*S*S+S*S*inC*outC



Inception

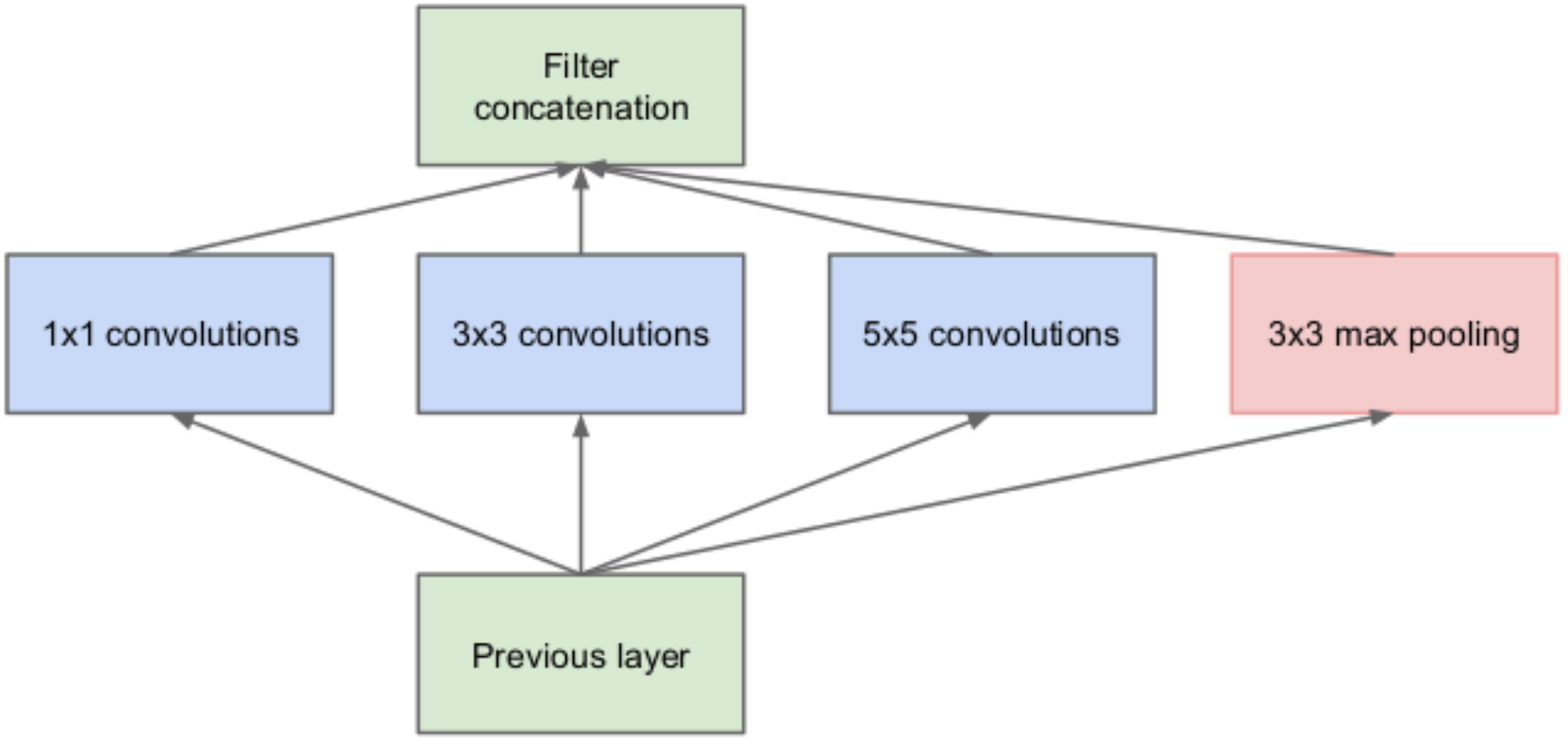


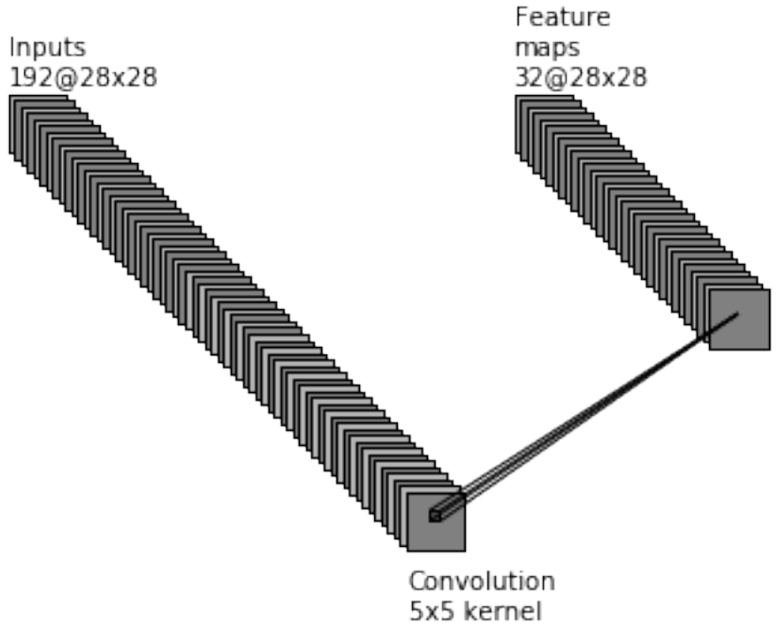
- GoogleNet architecture with 9 {\it inception} modules

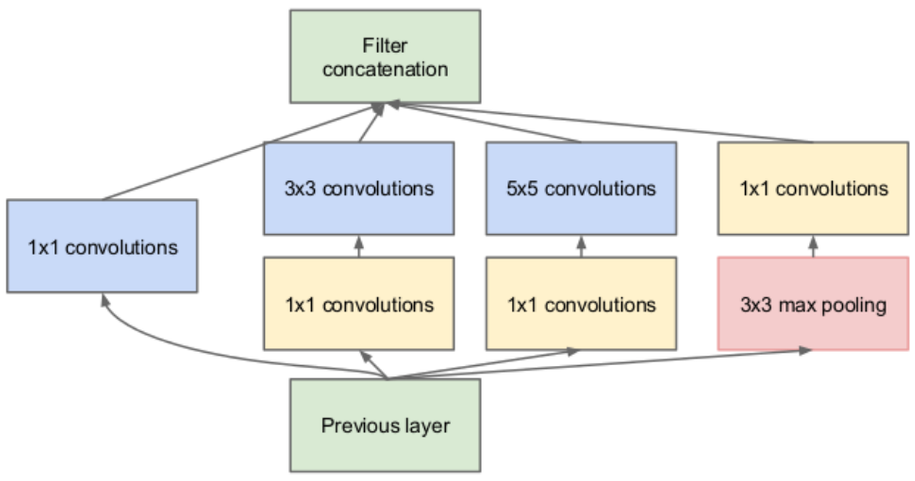
- Q: What is the correct filter size to use? 3x3, 5x5, 1x1?

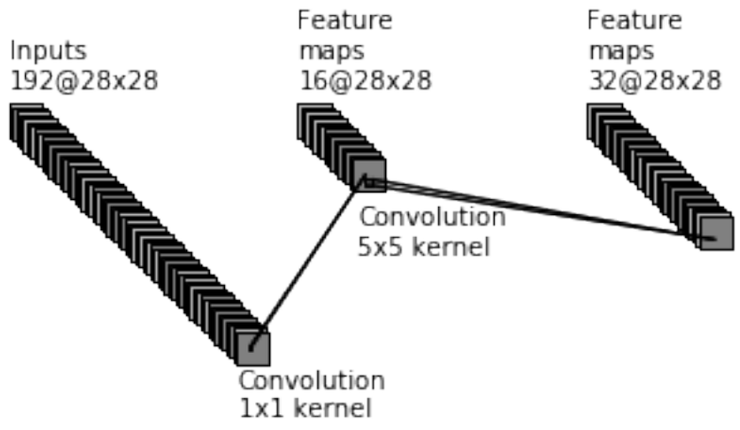
- A: Why not all of them?

Implementation





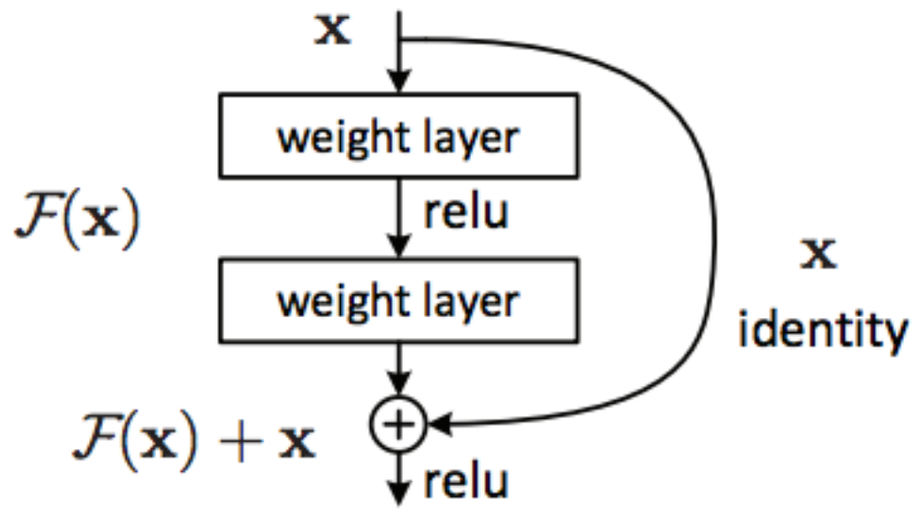




Resnet

- With network depth increasing, accuracy gets saturated (which might be unsurprising) and then *degrades rapidly*. Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error.
    - Why don’t the higher layers just learn an identity function?
    - Then this deeper network can not have higher training error than shallow network.
    - (Higher training error is mainly due to vanishing gradients)

Skip Connection



- hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers

- block is trying to model is closer to an identity mapping than to a zero mapping, and that it should be easier to find the perturbations with reference to an identity mapping than to a zero mapping. This simplifies the optimization of our network at almost no cost. Subsequent blocks in our network are thus responsible for fine-tuning the output of a previous block, instead of having to generate the desired output from scratch.

- Also gradients can flow through the identity branch