State of the art architectures in classification for vision

Alexander Binder

University of Oslo (UiO)

February 23, 2023



UiO : Department of Informatics
University of Oslo

Takeaway points

at the end of this lecture you should be able to summarize:

- dropout and batchnorm
- VGG – specialties
- googlenet – specialties
- resnet – specialties
- densenet – specialties
- networks used for segmentation and GANs, image captioning and other tasks use similar building blocks

Takeaway points

good practices in state of the art models (not only for vision!!!):

- Batchnormalization or Layernorm
- residual connections/skip connections
- use an ensemble of models rather than a single model
- stack smaller kernels rather than use a big kernel – partially weakened in ConvNext (kernels up to size 7). The explanations for why these work well are often rather conceptual (except for ensembles).

Takeaway for this lesson:

- finetuning means you load weights from another pretrained model as much as you can
- load weights bottom-up until you find a layer where you cannot load weights
- Avoid to train from scratch whenever possible![a] Finetuning can improve performance when training with small sample sizes greatly as compared to training from scratch with a random initialization.

[a] in ML exceptions always apply

Takeaway for this lesson:

- finetuning can be used for models with different types of inputs and multiple forward streams, e.g. image and text – but always bottom up: from one of the inputs until the first layer where weights cannot be loaded anymore (bcs one changed the network design at this point to either a completely different layer, or due to shape mismatch). after one such blocker-layer, it makes no sense to load weights further above
- finetuning has three flavours: train all layers, train only the top layer, train only the k top-most layers

http://d2l.ai/chapter_convolutional-modern/index.html

Architecture: VGG

Simonyan & Zisserman, ICLR 2015

https://arxiv.org/abs/1409.1556

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

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

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

- contribution: old-style network: repeated blocks of: (convolution-relu)^{*n}-pooling
- only 3x3-convolutions to achieve larger fields of view by stacking
- very large number of parameters: 130 millions!
- 3 fully connected layers contain a large part of the parameters
- dropout layer for less overfitting between fc layers
- 2014 ILSVRC competition second place.

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

- Has one parameter: keep or drop probability p.
- At test time, rescaled identity.
- At training time: set randomly $1 - p$ of all neurons to zero. A way of adding noise to the learning problem.

why does it work?

Why does this work?

- two dimensions of the feature map $\phi_{d_1}(x)$ and $\phi_{d_2}(x)$ may have a correlation which helps to classify sample x on training data

  example: 95% of all the time we have on training data for a pair (x, y) of sample and label:

  $2\phi_{d_1}(x) - \phi_{d_2}(x) > 0$ whenever the label $y > 0$,
  but **if** this correlation would not be not present in test data – picking up such a correlation results in overfitting.
- setting $\phi_{d_1}(x)$ or $\phi_{d_2}(x)$ to zero, prevents the algorithm from setting weights to use this non-generalizing correlation in a too strong way. it has to rely also on other correlations in the data.

One way to explain is:

- noise via setting features to zero reduces statistical correlation between features.
- Then the model cannot overfocus on one single correlation
- instead it has to rely on a mixture of several different correlations between features. Some of them may not generalize / hold true for the source of your data P_{test}.

Modern applications of Dropout: model-based uncertainty estimates

- To compute estimates of model-based uncertainty of the model when trainable parameters are drawn from a distribution. What?
- idea: logistic regression outputs had a probabilistic interpretation $p(Y|X = x)$ for input vectors x.
- closer look: this probability also depends on the trainable model parameters: $P(Y|X = x, \theta)$. For logistic regression: $y = \frac{1}{1+\exp(-z)}$, $z = w \cdot x + b = P(Y = 1|X = x, \theta = (w, b))$

- The output probability in logistic regression also depends on the trainable model parameters: $P(Y|X = x, \theta)$:
  $y = \frac{1}{1+\exp(-z)}$, $z = w \cdot x + b = P(Y = 1|X = x, \theta = (w, b))$
- Same holds for softmax outputs. The k-th output models $P(Y = y_k|X = x, \theta)$ – the probability for the k-th class. θ are all your trainable NN weights (conv-layers, dense layers, all other layers with trainable parameters.)

in general: can view the predictions Y of a model as a probabilistic function of inputs X and parameters θ: $P(Y|X, \theta)$

This does not give yet model-based uncertainty.

Lets look at the training process with training data $D_{tr} = \{(x_i, y_i), 1 \leq i \leq N\}$. Training can be seen as a mapping $(\cdot) \mapsto (\cdot)$:

$$(D_{tr}, \theta_{init}, \text{seeds of RNG}, SGD(RNG), \text{Data augmentation}(RNG)) \mapsto \theta$$

given fixed training data D_{tr} the outcome of parameters depends on randomness in the training! SGD, data augmentation (next lecture).

in general: can view the parameters θ of a trained model as a probabilistic function of the training data D_{tr}: $P(\theta|D_{tr})$

If we knew (no we dont know!) $P(\theta|D_{tr})$, then we could integrate over this uncertainty in model parameters:

$$P_*(Y|X, D_{tr}) = \int_\theta P(Y|X, \theta)P(\theta|D_{tr})d\theta$$

here: X is the test input. D_{tr} is the training data

If we knew $P(\theta|D_{tr})$, then we could integrate over this uncertainty in model parameters:

$$P_*(Y|X, D_{tr}) = \int_\theta P(Y|X, \theta)P(\theta|D_{tr})d\theta$$

Could use this then to compute the variance of Y conditioned on X, D_{tr}:

$$Var(Y|X, D_{tr}) = E_{y\sim P_*(Y|X,D_{tr})}[Y^2] - (E_{y\sim P_*(Y|X,D_{tr})}[Y]^2)$$

- Conditional variance, because y drawn from a conditional probability $P_*(Y|X, D_{tr})$.
- $P(\theta|D_{tr})$ is the model of the probability of parameters given training data and the randomness in the training
  $(D_{tr}, \theta_{init}, \text{seeds of RNG}, SGD(RNG), \text{Data augment}(RNG)) \mapsto \theta$
- $P_*(Y|X, D_{tr})$ is computed from $P(\theta|D_{tr})$ and the prediction model $P(Y|X, \theta)$ conditioned on training parameters θ

How does dropout come into play ?

- We dont know $P(\theta|D_{tr})$
- use dropout to set trained weights θ to zero in your fully trained model. This defines a probability distribution $q(\theta|\gamma)$ over model weights θ which depends on drop-probabilities γ.
- goal optimize γ so that this approximation holds:

$$P(\theta|D_{tr}) \overset{!!}{\approx} q(\theta|\gamma)$$

How to do that ? We need to know how to compute $P(\theta|D_{tr})$! Not treated in this lecture: how to compute $P(\theta|D_{tr})$. How to optimize for γ: evidence lower bound (ELBO). See backup slides for those who want to know

Modern applications of Dropout: model-based uncertainty estimates

- faces also criticisms: the truthfulness to Bayesian modelling is questioned:
  - uncertainty is not equivalent to outlier detection capabilities: Liu et al. https://arxiv.org/abs/2112.05000
  - Le Folgoc et al. https://arxiv.org/abs/2110.04286

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

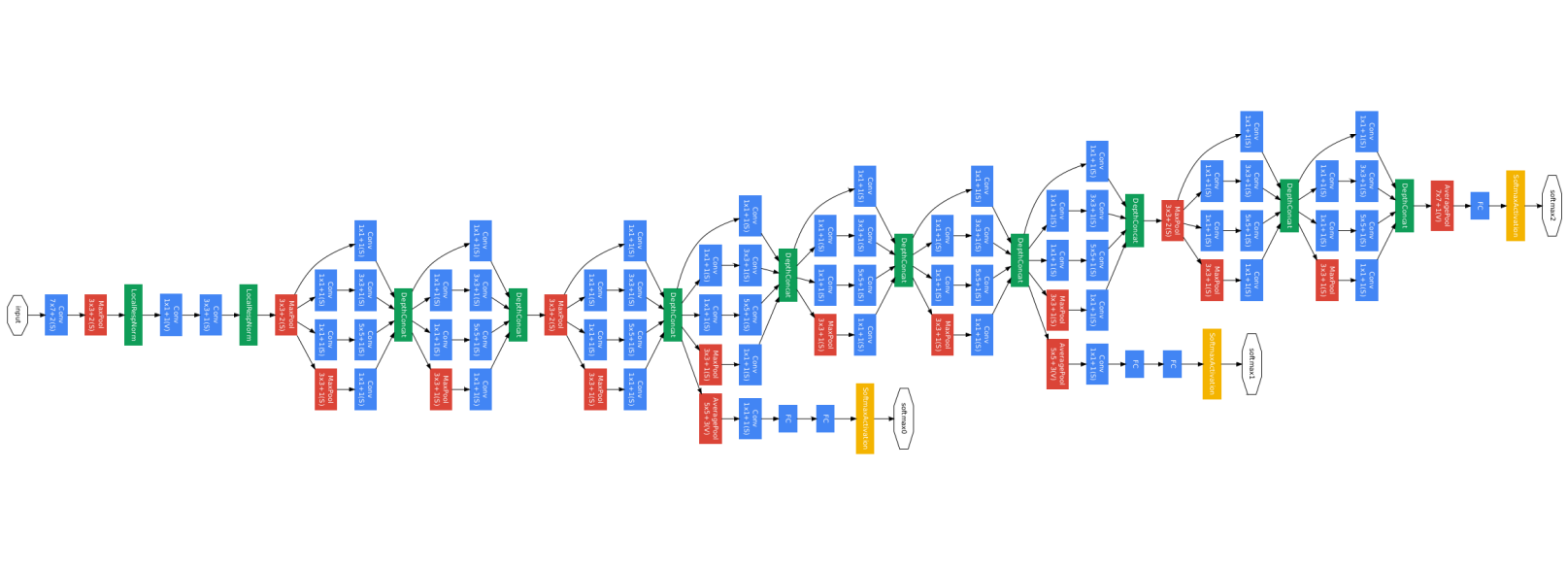
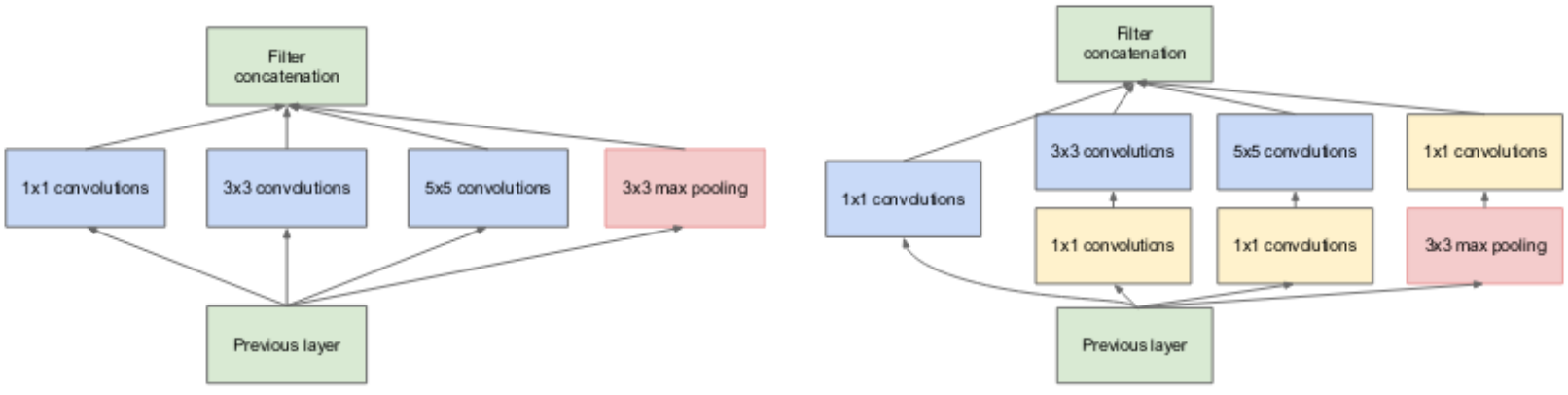


Figure 3: GoogLeNet network with all the bells and whistles

https://arxiv.org/abs/1409.4842

- contribution1: auxiliary output losses – at training time only – for injecting gradient flow in layers close to the bottom. Auxiliary output is a separate classification output. + cross entropy loss attached for training. Loss to be optimized is weighted sum of main loss and aux losses.

- contribution2: inception module: convolution layers in parallel with different effective kernel sizes – this is classical multi-scale processing



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

- notable: 1x1 convolutions to reduce parameters

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

- contribution3: at test time: average over multiple classifiers and massive data augmentation

Number of models	Number of Crops	Cost	Top-5 error	compared to base
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Table 3: GoogLeNet classification performance break down

preview for data augmentation:

http://d2l.ai/chapter_computer-vision/image-augmentation.html

- many small finetuning ideas on top of v1 architecture
  - with batchnorm (big gain)
  - 1 auxiliary loss only with BN (bcs of better flow due batchnorm!)
  - no 7×7 filters at the start of the net refactored them in to 3 layers of 3×3
  - top-layers: modules with factorizations replacing 5×5 and 7×7 kernels
- commonly used instead of googlenet
- no residual connections (introduced in Inception v4)

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Fig 5,6,7 from table on the last slide. Remember from here only the idea to replace larger $n \times n$ kernels by stacking 3×3 or (or layers composed of a pair $n \times 1$, $1 \times n$).

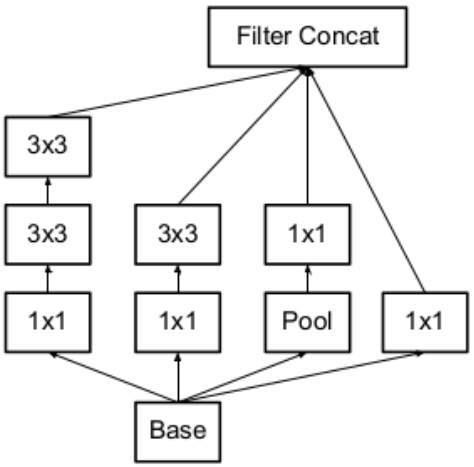


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

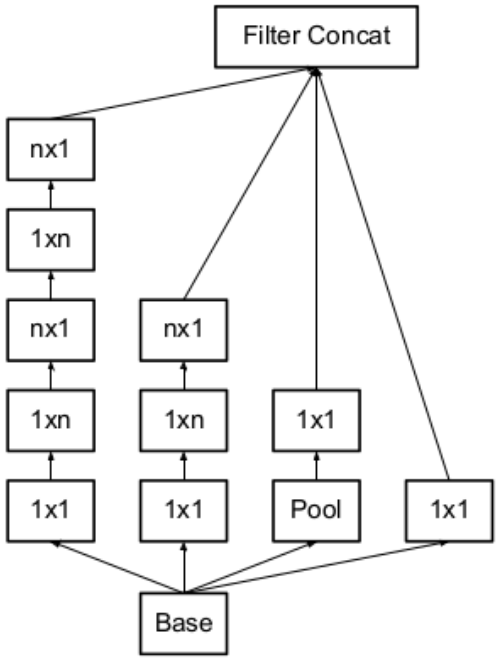


Figure 6. Inception modules after the factorization of the $n \times n$ convolutions. In our proposed architecture, we chose $n = 7$ for the 17×17 grid. (The filter sizes are picked using principle 3)

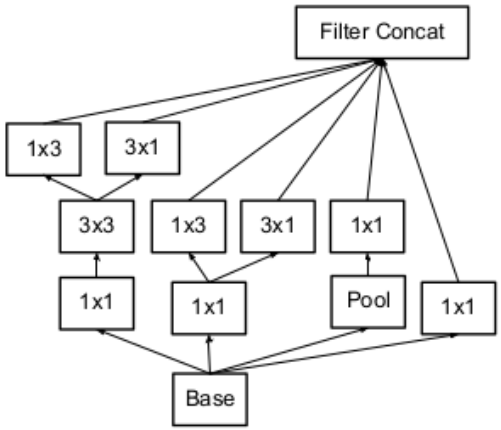
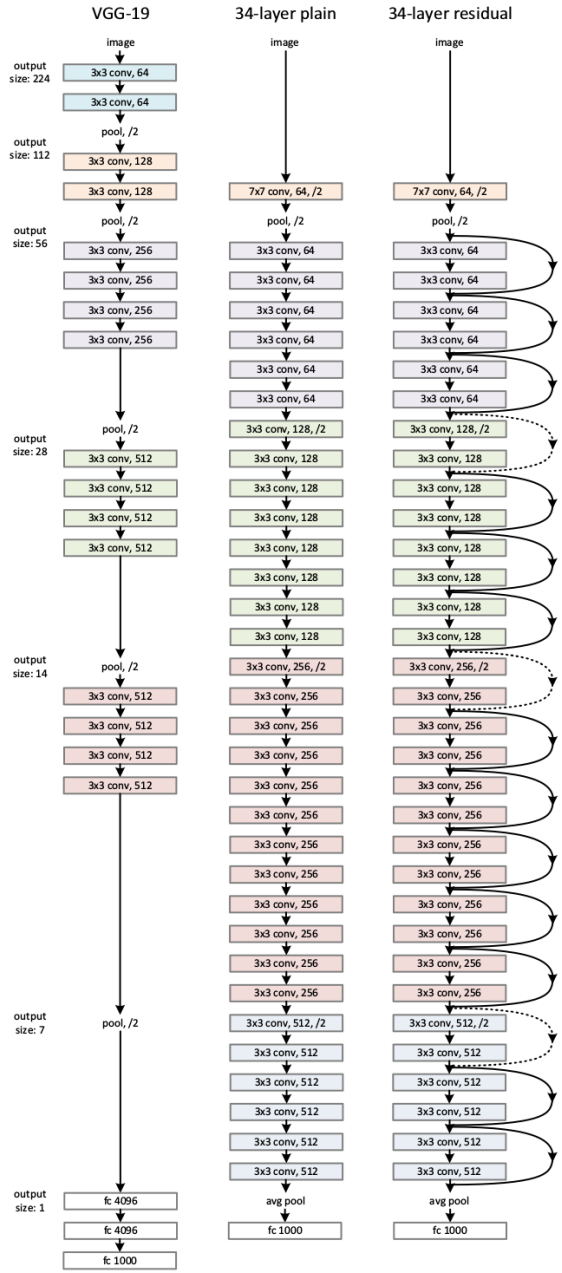


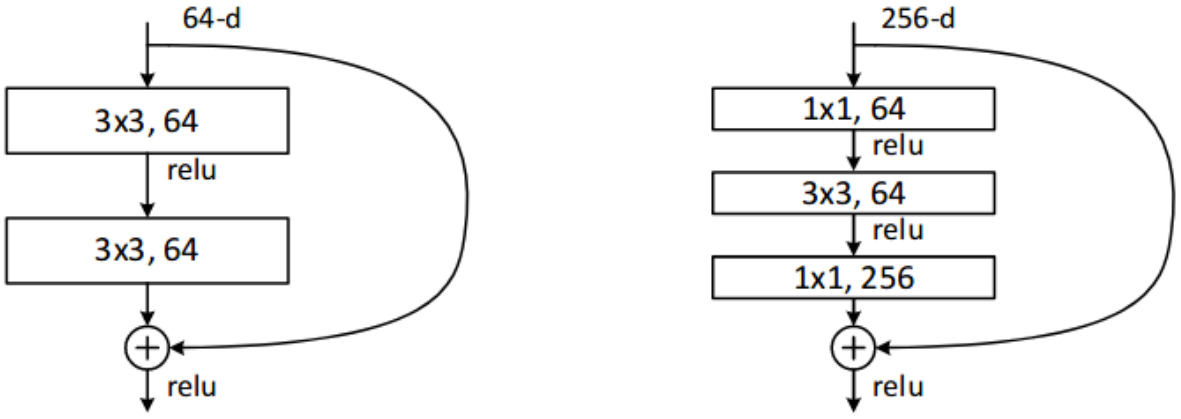
Figure 7. Inception modules with expanded the filter bank outputs. This architecture is used on the coarsest (8×8) grids to promote high dimensional representations, as suggested by principle 2 of Section 2. We are using this solution only on the coarsest grid, since that is the place where producing high dimensional sparse representation is the most critical as the ratio of local processing (by 1×1 convolutions) is increased compared to the spatial aggregation.

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

ResNets and residual connections

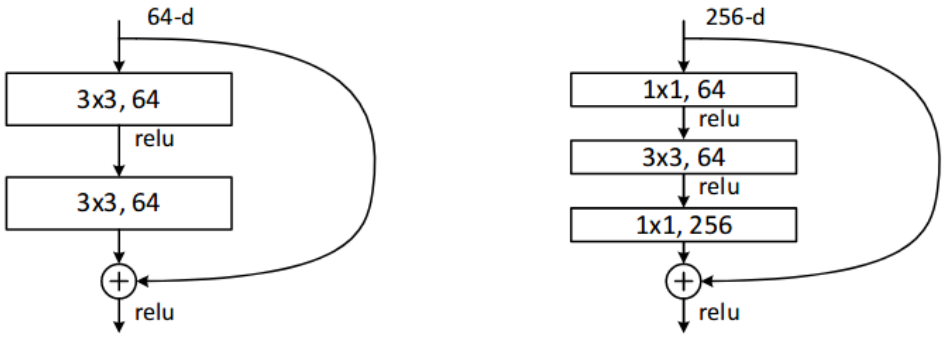




residual connection (2 conv blocks): $f(x) = x + C_1(C_2(x))$

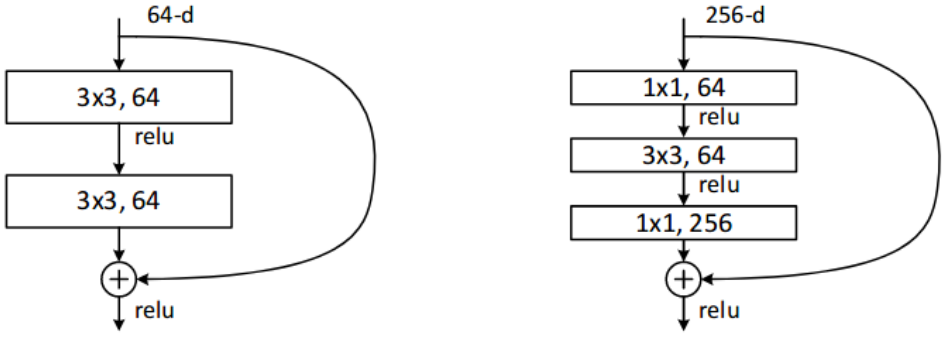
Why do residual connections work ?

- gradient flows as the identity through the shortcut, no vanishing gradient problem
- shortcut in forward pass: inputs feature from previous layer, convolutions across the parallel path can learn additionally non-linear function on top



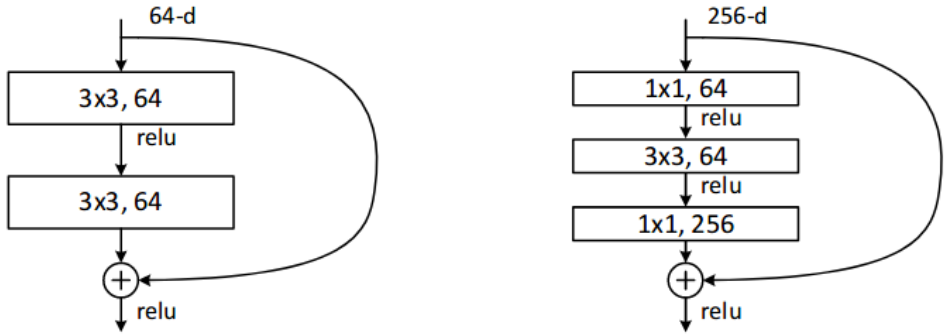
- identity+ optional nonlinearity: residual $\rightarrow$ convolutions across the parallel path can learn additionally non-linear function on top.
- option for quick unlearning to identity: if poor fit was learned during early phases of training, it can be undone: update weights of convolution layers to zero, then have only the identity.

  (unhindered information flow fwd & bwd)
- identity+ optional nonlinearity: allows to learn a more complex representation layer by layer: Network can start as: 1 conv layer and one fully connected layer, and (almost) only shortcuts in between. Convolution kernels can add layer by layer some nonlinearities.



Why do residual connections work ?

- gradient flows as the identity through the shortcut, no vanishing gradient problem
- identity+ optional nonlinearity: residual $\rightarrow$ convolutions across the parallel path can learn additionally non-linear function on top.
- later on: compare to the memory cell in LSTM (1998). Earlier idea for gradient flow through time steps.



- Resnets use also batchnormalization after every convolution before the ReLU

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

Ioffe and Szegedy, 2015 https://arxiv.org/pdf/1502.03167.pdf

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batchnorm at training time performs 2 steps

- step 1: take one neuron , normalize its outputs so that – over the all elements in your minibatch have zero mean and standard deviation 1
- step 2: apply a simple affine transformation on the normalized output $y = \gamma\hat{x} + \beta$
- after this output has standard deviation γ and mean β , γ and β trainable

Ioffe and Szegedy, 2015 https://arxiv.org/pdf/1502.03167.pdf

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batchnorm at training time performs 2 steps

- batchnorm at training time learns to output values which have constant mean and constant standard deviation (computed over the elements in a minibatch)
- convolution layers: compute mean, standard deviation, *a*, *b* not for one neuron and all samples in the minibatch but for all elements in the feature map of one channel in a conv layer and all samples in the minibatch – reduces parameters, treats each neuron in the same channel in the same way

Ioffe and Szegedy, 2015 https://arxiv.org/pdf/1502.03167.pdf

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batchnorm at training time performs 2 steps

- batchnorm at training time learns to output values which have constant mean and constant standard deviation (computed over the elements in a minibatch)
- training time: update running mean and running variance for your training dataset (will be needed for test time)
- in order to work well requires usually a batchsize of 8 at least, better 16 or 32 or more.

Batchnorm at val/test time performs 2 steps:

- step 1: take one neuron, normalize its outputs by the running mean and running variance learnt at training time.

$$\hat{x} = \frac{x - \mu_{run}}{\sqrt{\sigma^2_{run} + \epsilon}}$$

- step 2: apply $y = a\hat{x} + b$. a,b the learnt rescaling parameters
- meaning: this simulates that the test sample would come from a very large batch with mean and variance statistics equal to the training data. under this assumption, any synthetic minibatch composed of many test samples would have mean b and std deviation a.
- important/error source in coding: use `model.eval()` or `model.train(False)` at testing time for your neural network!

Why does batchnorm improve performance?
Equation on page 5 in the paper – Gradient with respect to inputs does not depend on scale of weight parameter anymore. Makes gradient flow more uniform across the channels of a layer.

Layer normalization

- Normalize for each sample separately. As if one would train with a batch-norm of 1
- used in SwinTransformers and ConvNext

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

Huang, Liu, van der Maaten, Weinberger,
https://arxiv.org/abs/1608.06993

- resnets to the extreme: within a block of same feature map size ("dense block"), each layer contains the feature maps of each previous layer (of the same block) via concatenation of feature maps.

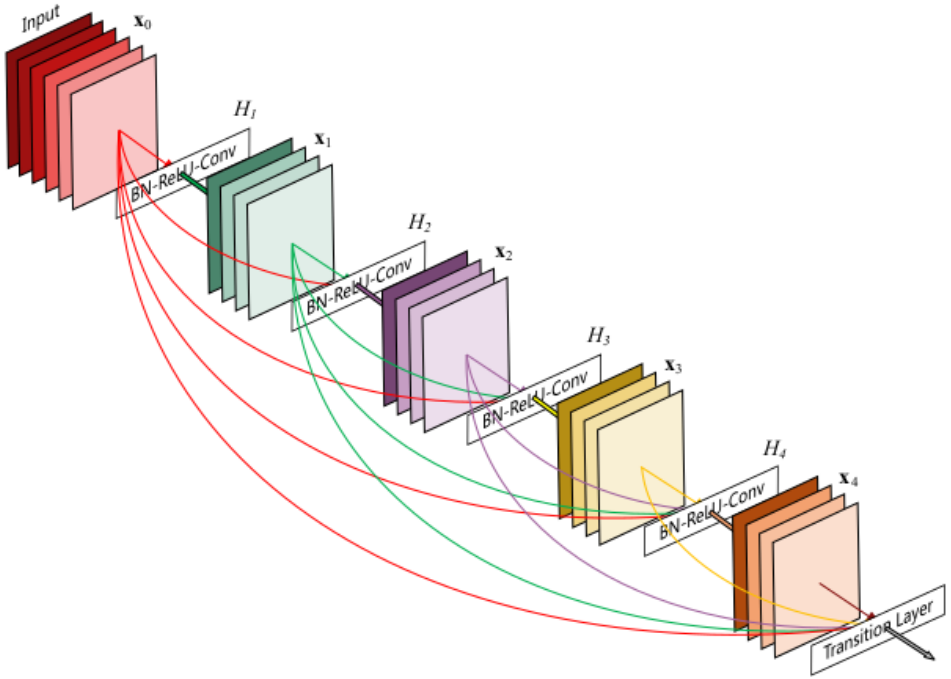


Figure 1: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input.

The whole net looks like:

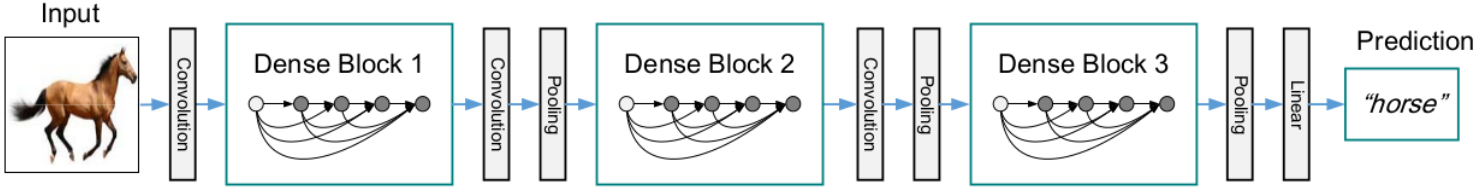


Figure 2: A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature-map sizes via convolution and pooling.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Table 1: DenseNet architectures for ImageNet. The growth rate for all the networks is $k = 32$. Note that each "conv" layer shown in the table corresponds the sequence BN-ReLU-Conv.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Table 1: DenseNet architectures for ImageNet. The growth rate for all the networks is $k = 32$. Note that each "conv" layer shown in the table corresponds the sequence BN-ReLU-Conv.

- Important parameter: growth rate - the number of **newly added** output channels in a convolution layer, typically small like 12,24,40
- problem: within a block that starts with k_0 channels, at depth index l one has as inputs $k_0 + (l-1) \cdot \textit{growthrate}$ many channels.
- Densenet-B: 1x1 convolutions with BN and ReLU before each layer (with $4 \cdot \textit{growthrate}$ output channels) to reduce the # of channels and parameters in subsequent 3x3 kernels.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

Table 1: DenseNet architectures for ImageNet. The growth rate for all the networks is $k = 32$. Note that each "conv" layer shown in the table corresponds the sequence BN-ReLU-Conv.

- Densenet-C: in transition layer (see figure: where feature map size is downscaled by 1/2): 1x1 conv generates as output channels half the number of incoming channels
- usually used: Densenet-BC

- usually used: Densenet-BC
- low parameter count among the heavier networks, good performance

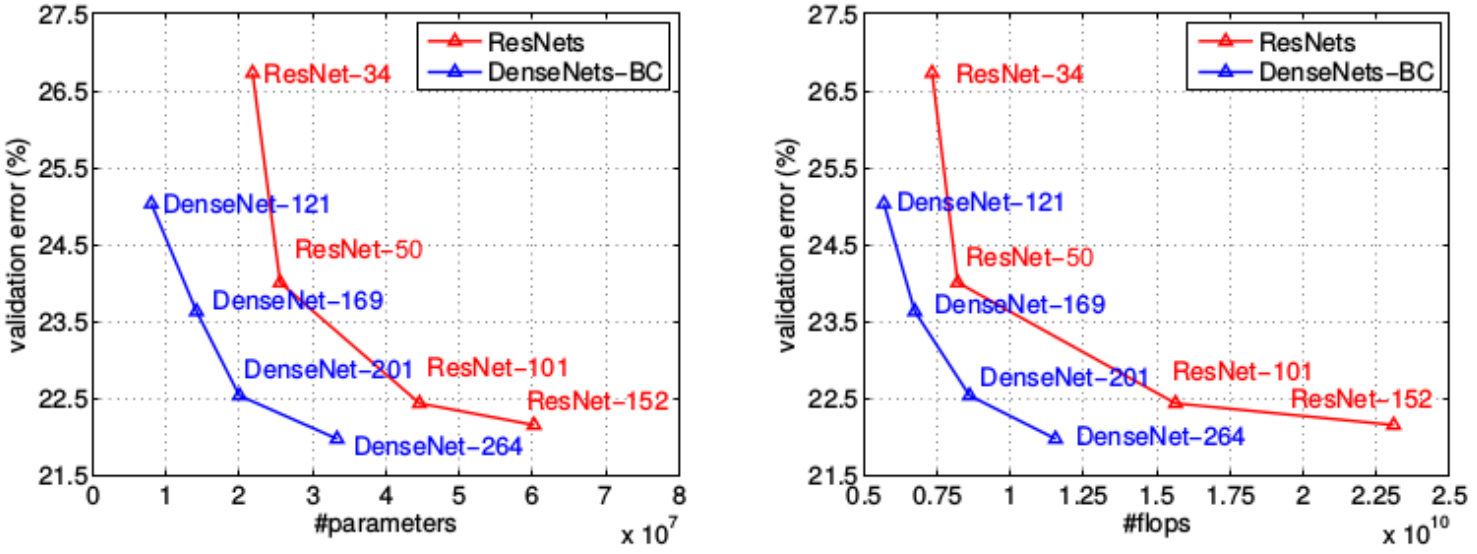


Figure 3: Comparison of the DenseNets and ResNets top-1 error rates (single-crop testing) on the ImageNet validation dataset as a function of learned parameters (*left*) and FLOPs during test-time (*right*).

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

The main lesson for deep learning

Do not train deep neural networks from scratch![a] Always initialize the NN with weights from similar tasks trained on a very large dataset, except your data is in the order of hundred thousands and more.

[a] ML has always exceptions :)

While for some tasks with precise knowledge training from scratch works, 99% fine tuning of all layers is better than training from scratch.
where to get and how ? `torchvision.models`
https://pytorch.org/docs/stable/torchvision/models.html

Mnist and Cifar-10 work without finetuning - are misleading.
Note the simplicity of the tasks: images with 28×28, or 32×32 have limited variability and complexity compared to larger images!
Mnist and Cifar-10 are very useful for testing small ideas, but they are outliers within deep learning tasks.

Practice session: you will take a deep network (densenet or a mobilenet), initialize it with weights from a 1000 class imagenet task, and then retrain it for 102 flowers classes. Why one can re-use weights from 1000 object classes that are mostly things and animals for flowers? The low level filters likely will be very similar.

- What needs to be changed? The last layer: to the number of output classes in your problem instead of 1000.
- therefore: last layer will not use pretrained weights
- see Figure 14.2.1 in https://d2l.ai/d2l-en.pdf in Chapter 14

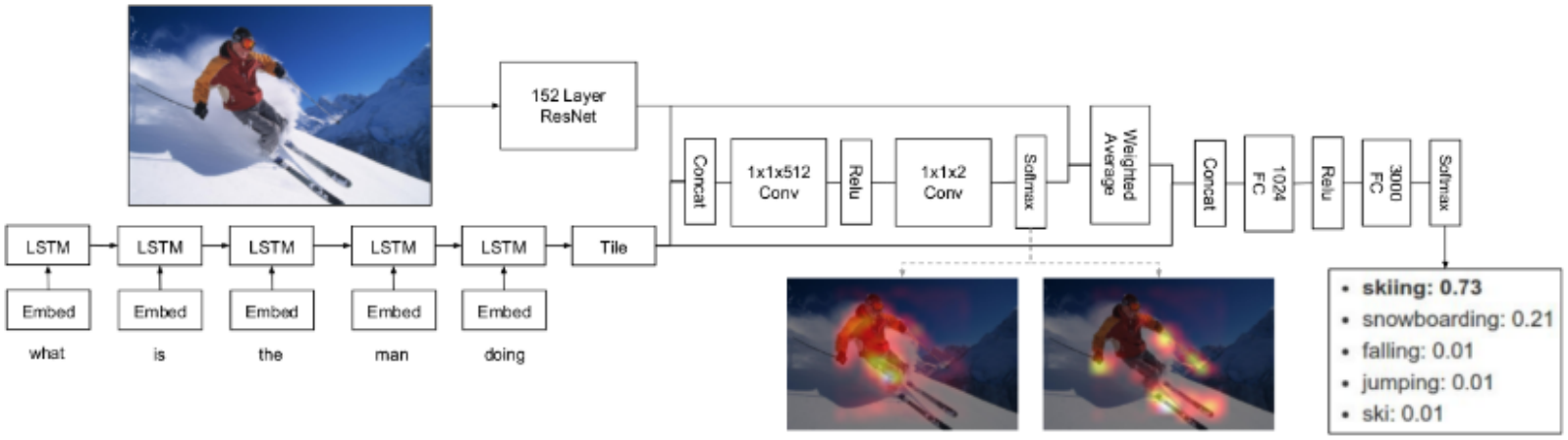


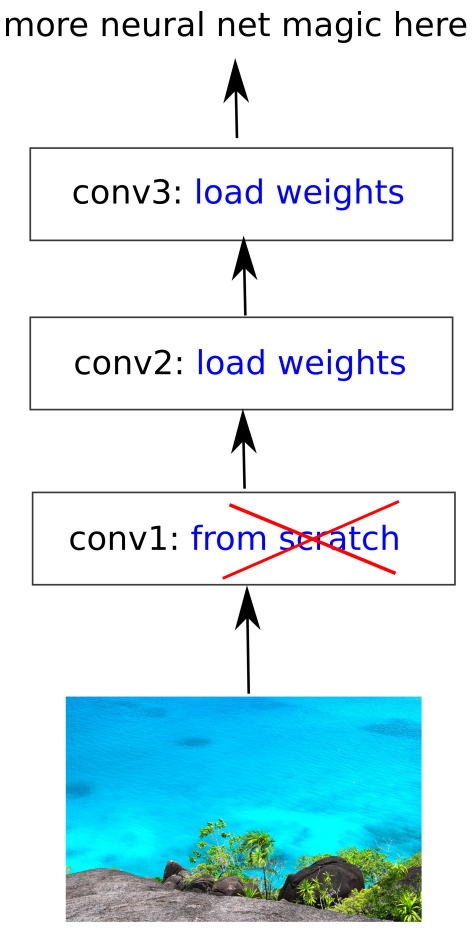
Figure 2. An overview of our model. We use a convolutional neural network based on ResNet [9] to embed the image. The input question is tokenized and embedded and fed to a multi-layer LSTM. The concatenated image features and the final state of LSTMs are then used to compute multiple attention distributions over image features. The concatenated image feature glimpses and the state of the LSTM is fed to two fully connected layers two produce probabilities over answer classes.

Kazemi and Elqursh https://arxiv.org/pdf/1704.03162.pdf

What parts here can profit from transfer learning?

Above shows a VQA-architecture with attention. An image is processed by a CNN. A question is processed by embeddings, then an LSTM. The features are fused and weighted by attention layers. Final prediction is made by FC-layers with classification over possible answers as output.

It makes no sense to load weights for a layer, when one skips loading weights for any layer below. why ?



a neural net where finetuning makes NO sense –because we skip a layer early on.

Why does fine tuning help?

Alias	Network	# Parameters	Top-1 Accuracy	Top-5 Accuracy	Origin
alexnet	AlexNet	61,100,840	0.5492	0.7803	Converted from pytorch vision
densenet121	DenseNet-121	8,062,504	0.7497	0.9225	Converted from pytorch vision
densenet161	DenseNet-161	28,900,936	0.7770	0.9380	Converted from pytorch vision
densenet169	DenseNet-169	14,307,880	0.7617	0.9317	Converted from pytorch vision
densenet201	DenseNet-201	20,242,984	0.7732	0.9362	Converted from pytorch vision
inceptionv3	Inception V3 299x299	23,869,000	0.7755	0.9364	Converted from pytorch vision
mobilenet0.25	MobileNet 0.25	475,544	0.5185	0.7608	Trained with script
mobilenet0.5	MobileNet 0.5	1,342,536	0.6307	0.8475	Trained with script
mobilenet0.75	MobileNet 0.75	2,601,976	0.6738	0.8782	Trained with script
mobilenet1.0	MobileNet 1.0	4,253,864	0.7105	0.9006	Trained with script
mobilenetv2_1.0	MobileNetV2 1.0	3,539,136	0.7192	0.9056	Trained with script
mobilenetv2_0.75	MobileNetV2 0.75	2,653,864	0.6961	0.8895	Trained with script
mobilenetv2_0.5	MobileNetV2 0.5	1,983,104	0.6449	0.8547	Trained with script
mobilenetv2_0.25	MobileNetV2 0.25	1,526,856	0.5074	0.7456	Trained with script

Consider training a neural network: non-convex problem: One always finds some local optimum. Quality of it varies.

Deep NNs: high dimensionality of their parameters.

https://mxnet.incubator.apache.org/api/python/gluon/model_zoo.html .

Training 15 million parameters with 1000 samples violates the golden rule. You will overfit for sure.

Why does fine tuning help?

Alias	Network	# Parameters	Top-1 Accuracy	Top-5 Accuracy	Origin
alexnet	AlexNet	61,100,840	0.5492	0.7803	Converted from pytorch vision
densenet121	DenseNet-121	8,062,504	0.7497	0.9225	Converted from pytorch vision
densenet161	DenseNet-161	28,900,936	0.7770	0.9380	Converted from pytorch vision
densenet169	DenseNet-169	14,307,880	0.7617	0.9317	Converted from pytorch vision
densenet201	DenseNet-201	20,242,984	0.7732	0.9362	Converted from pytorch vision
inceptionv3	Inception V3 299x299	23,869,000	0.7755	0.9364	Converted from pytorch vision
mobilenet0.25	MobileNet 0.25	475,544	0.5185	0.7608	Trained with script
mobilenet0.5	MobileNet 0.5	1,342,536	0.6307	0.8475	Trained with script
mobilenet0.75	MobileNet 0.75	2,601,976	0.6738	0.8782	Trained with script
mobilenet1.0	MobileNet 1.0	4,253,864	0.7105	0.9006	Trained with script
mobilenetv2_1.0	MobileNetV2 1.0	3,539,136	0.7192	0.9056	Trained with script
mobilenetv2_0.75	MobileNetV2 0.75	2,653,864	0.6961	0.8895	Trained with script
mobilenetv2_0.5	MobileNetV2 0.5	1,983,104	0.6449	0.8547	Trained with script
mobilenetv2_0.25	MobileNetV2 0.25	1,526,856	0.5074	0.7456	Trained with script

You can learn filters well only when you have enough training samples, often one needs hundreds of thousands.

Non-convex problem: optimum depends on initialization.

When having only a few thousand samples it is best to start from a good initialization – loading weights does that.

But why it is a good initialization?

Why does fine tuning help?

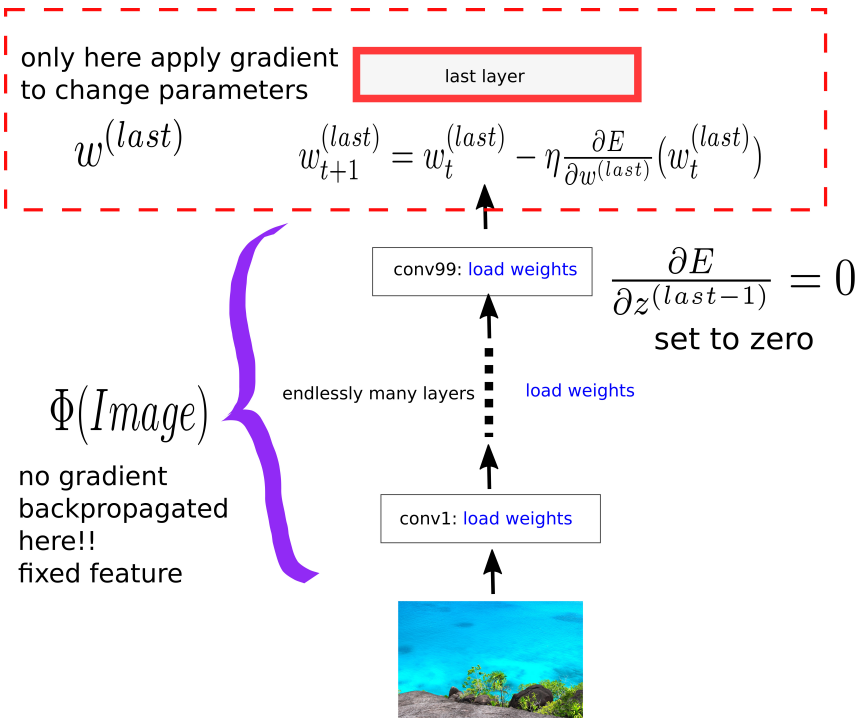
- Finetuning preinitializes your network to some features which were good on another task.
- empirical evidence: low-level features in deep networks learnt over wide and general tasks (e.g. Imagenet) can be reused for many other tasks, even with strange color distributions or geometrical tasks

Why does fine tuning help?

- A good initialization from finetuning will be destroyed when trained too long with too little samples.
- In practice backpropagating gradients changes the highest level weights faster (due to vanishing gradients), so that – at the beginning of training – the weights in the upper layers adapt faster towards what one wants to learn – and the overfitting by changing lower layer weights to bad optima sets in only later.

A good initialization from finetuning will be destroyed when trained too long... ?

- Finetuning has a special case: when the number of training data is very small, then one may want to retrain only the top layers. Finetuning in the narrowest sense: only train the top-layer. Works best when the number of training samples is very small.



Here an example when you retrain only the last layer as an extreme case of finetuning. This is often shown in tutorials

training only top layers:

- can be better for very small data sizes
- for larger data sizes training all layers can be better ... check on validation data
- training only top-layers: without data augmentation can precompute bottom features for a speed up (but usually data augmentation improves test error! ... tradeoff speed vs performance)

Outline

1. Dropout
2. Googlenet v1 / Inception v1
3. ResNets and residual connections
4. Batch normalization
5. DenseNets
6. Finetuning
7. SOA 2023?

in later lectures:

- Swin Transformers https://arxiv.org/abs/2103.14030 , Swin Transformers V2 https://arxiv.org/abs/2111.09883
- ConvNext: https://arxiv.org/abs/2201.03545 – takes inspiration from Transformer network design
- but also: Efficientnet https://arxiv.org/abs/1905.11946
- notable: normalizer-free nets https://arxiv.org/abs/2102.06171

in the next lectures ...

You do not need to memorize the following architectural details of a ConvNext.

Table 5 and Table 6 for training and finetuning settings, Table 9 for architectural details.

- changing the first convolution (from (7,7) stride 2 with maxpool stride 2) to (4,4) stride 4
- use depth-wise separable convolutions with more channels instead - mixes either channels or spatial dimensions, but not both
- inverted bottleneck block with larger kernel size (7×7) for the separable convolution: See Fig 4 in https://arxiv.org/abs/2201.03545
- give up BatchNorm-ReLU pairs
- ReLU $\rightarrow$ GELU https://pytorch.org/docs/stable/generated/torch.nn.GELU.html, use it only between the two 1×1 Convolutions

Table 5 and Table 6 for training and finetuning settings, Table 9 for architectural details.

- Batchnorm $\rightarrow$ Layernorm, use it inside a convnext residual block only after the separable convolution.
- Layernorm also after the stem, before each down-sampling layer (2×2-conv stride 2), one after the global average pooling at the end. Idea: layernorm added whenever resolution changes
- linear warmup of the learning rate over 20 epochs

Table 5 and Table 6 for training and finetuning settings, Table 9 for architectural details.

- StochasticDepth (omit to use residual blocks with a certain probability)
- LabelSmoothing (a mixture of the one-hot ground-truth label distribution with a small amount of a fixed label-agnostic distribution like the uniform one),
- Layer-Scale at the end of a residual block (trainable per channel weighting)
- HEAVY DataAugmentation

Questions?!

The next slides are not part of the lecture material. only for interested students.

- We need to know how to compute $P(\theta|D_{tr})$!

$$P(\theta|D_{tr}) = \frac{p(D_{tr}|\theta)p_{prior}(\theta)}{\int_\theta p(D_{tr}|\theta)p_{prior}(\theta)d\theta}$$

$\int_\theta p(D_{tr}|\theta)p_{prior}(\theta)d\theta$ does not depend on θ. Therefore this normalizer does not need to be included when we try to match $P(\theta|D_{tr}) \overset{!!}{\approx} q(\theta|\gamma)$ and we obtain:

$P(\theta|D_{tr}) = p(D_{tr}|\theta)p_{prior}(\theta) \cdot$ some const > 0 **not** dependent on θ

$P(\theta|D_{tr}) \propto p(D_{tr}|\theta)p_{prior}(\theta)$ as common notation in literature

- goal optimize γ so that this approximation holds:

$$P(\theta|D_{tr}) \overset{!!}{\approx} q(\theta|\gamma)$$

How to do that ? We know:

$$P(\theta|D_{tr}) = \frac{p(D_{tr}|\theta)p_{prior}(\theta)}{\int_\theta p(D_{tr}|\theta)p_{prior}(\theta)d\theta}$$

$$P(\theta|D_{tr}) = p(D_{tr}|\theta)p_{prior}(\theta) \cdot \text{ some const not dep on } \theta$$

- $p_{prior}(\theta)$ is your prior over parameters. Usually known from probabilistic initialization of the neural net
- By definition of $p(D_{tr}|\theta)$ – these are the likelihood of your labels or your data[1] for fixed parameters θ:

$$p(D_{tr}|\theta) = \prod_{i:(x_i,y_i)\in D_{tr}} p(Y = y_i|X = x_i, \theta)$$

Therefore: $p(D_{tr}|\theta)$ can be obtained by plugging in the training data into the model!

[1] see three slides later for that case

Result:

$$P(\theta|D_{tr}) = \prod_{i:(x_i,y_i)\in D_{tr}} p(Y = y_i|X = x_i, \theta) p_{prior}(\theta) \cdot \text{ some C not dep on } \theta$$

- $p_{prior}(\theta)$ is your prior over parameters. Usually known from probabilistic initialization of the NN
- Result: $P(\theta|D_{tr})$ can be computed up to a constant which does not depend on θ
- Now can use any optimization to match $P(\theta|D_{tr}) \overset{!!}{\approx} q(\theta|\gamma)$

- Now can use any optimization to match $P(\theta|D_{tr}) \overset{!!}{\approx} q(\theta|\gamma)$
- common idea: choose γ to minimize $KL(q(\theta|\gamma)||P(\theta|D_{tr}))$

$$\text{argmin}_\gamma KL(q(\theta|\gamma)||P(\theta|D_{tr}))$$

$$KL(q(\theta|\gamma)||P(\theta|D_{tr})) = \int_\theta q(\theta|\gamma) \log \frac{q(\theta|\gamma)}{P(\theta|D_{tr})}$$

$$= \int_\theta q(\theta|\gamma) \log \frac{q(\theta|\gamma)}{\prod_{(x_i,y_i)\in D_{tr}} p(y_i|x_i,\theta) p_{prior}(\theta) C}$$

$$= \int_\theta q(\theta|\gamma)(\log \frac{q(\theta|\gamma)}{p_{prior}(\theta)} + \log \frac{1}{\prod_{(x_i,y_i)\in D_{tr}} p(y_i|x_i,\theta)} - \log C)$$

$$= -\log(C) + KL(q(\theta|\gamma)||p_{prior}(\theta)) - \int_\theta q(\theta|\gamma) \log \prod_{(x_i,y_i)\in D_{tr}} p(y_i|x_i,\theta)$$

side note: $P(D_{tr}|\theta)$ can have two meanings here, either

$$P(D_{tr}|\theta) = \prod_{i:(x_i,y_i)\in D_{tr}} p(Y = y_i|X = x_i, \theta) \text{ or}$$

$$P(D_{tr}|\theta) = \prod_{i:(x_i,y_i)\in D_{tr}} p(Y = y_i, X = x_i|\theta)$$

in the second case treatment is very similar:

$$p(Y = y_i, X = x_i|\theta) = p(Y = y_i|X = x_i, \theta)p(X = x_i|\theta)$$

but $p(X = x_i|\theta)$ is the density of the input data, and this is independent of trainable parameters θ. This implies: $p(X = x_i|\theta) = p(X = x_i)$. In that case:

$$p(Y = y_i, X = x_i|\theta) = p(Y = y_i|X = x_i, \theta)p(X = x_i)$$

where $p(X = x_i)$ is another positive constant not dependent on θ