Deep Learning for Visual Computing

Modern Classification Networks

Christopher Pramerdorfer
Computer Vision Lab, TU Wien

Image from twitter/mechanicmind_ai, created using midjourney

Topics

Optimization vs. Machine Learning

- Regularization

Modern classification networks

- Residual networks
- Efficient architectures

Pushing classification performance

- Fine-tuning
- Handling imbalanced data

Optimization vs. Machine Learning
Regularization

Recall from last lecture that our goal is to

- Reach high performance on unseen (validation/test) data
- By training on training data

For optimal results

- Networks should have enough capacity to overfit
- But be configured not to via regularization

The purpose of regularization is to

- Improve the validation/test performance
- At the possible expense of training performance

Usually by decreasing the models variance

- Sensitivity to small changes in training set
- And thus proneness to overfitting

A penalty on large weights is almost always used

- Prevents certain inputs from dominating output
- Encourages model to use all inputs

Biases are not critical

- Usually not subject to regularization

A common method is L2 regularization

- Implemented by adding regularization term to loss function

$$L_{reg}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \frac{\delta}{2}\|\mathbf{w}\|^2$$

$\mathbf{w} \subset \boldsymbol{\theta}$ is vector of all weights

$\delta \in (0, 1)$ controls amount of regularization

- Usually $\delta \in [0.0001, 0.01]$

Ignoring bias we get $\nabla L_{reg}(\mathbf{w}) = \nabla L(\mathbf{w}) + \delta\mathbf{w}$

- $\|\mathbf{w}\|^2 = \mathbf{w} \cdot \mathbf{w}$ so gradient is $2\mathbf{w}$ (product rule)

So gradient descent update becomes

$$\begin{aligned} \mathbf{w} &= \mathbf{w} - \alpha(\nabla L(\mathbf{w}) + \delta\mathbf{w}) \\ &= \mathbf{w} - \alpha\delta\mathbf{w} - \alpha\nabla L(\mathbf{w}) \\ &= (1 - \alpha\delta)\mathbf{w} - \alpha\nabla L(\mathbf{w}) \end{aligned}$$

Weights shrink by constant factor before each update

- So if $\nabla L(\mathbf{w}) = \mathbf{0}$, weights would decay towards $\mathbf{0}$
- Note that α also affects regularization

Decay strength δ is another hyperparameter

- No effect if too small
- Dominates data loss (e.g. cross-entropy) if too large

A very similar approach is weight decay

- Do not modify loss function
- Adapt weight update rule to subtract $\alpha\delta\mathbf{w}$

L2 regularization affects optimizer

- Leads to undesirable behavior with e.g. Adam
- So always use version of Adam with weight decay
- PyTorch: `torch.optim.AdamW`

Dropout [4] is a layer whose neurons

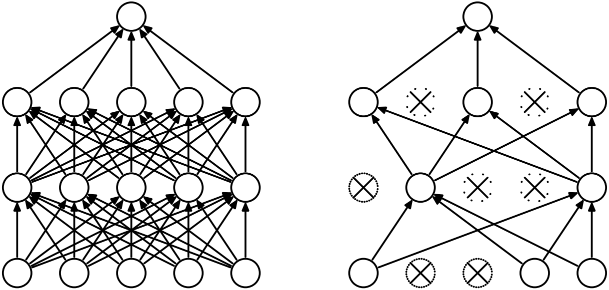
- Output 0 with probability p
- Forward input unchanged with probability $1 - p$
- But only during training

Usually placed before last (dense) layer

- Has effect of temporarily "dropping" neurons
- Because 0 does not change output

Not as popular anymore

Optimization vs. Machine Learning
Regularization – Dropout



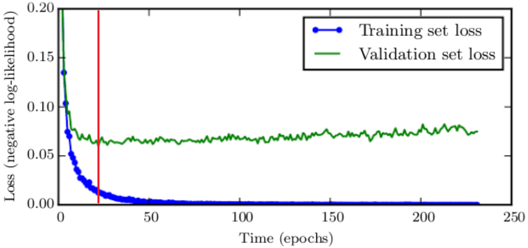
(a) Standard Neural Net　　(b) After applying dropout.

Image from [4]

Optimization vs. Machine Learning
Early Stopping

Early stopping aims to avoid overfitting by

- Storing a copy of $\boldsymbol{\theta}$ with best validation performance p
- Stopping if p does not improve anymore for e epochs
- Using the stored $\boldsymbol{\theta}$ afterwards



Optimization vs. Machine Learning
Suggestions

Data

- Use as much as you can get
- Always utilize (sensible) training data augmentation

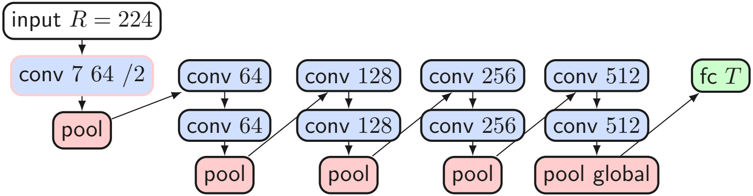
Regularization

- Always use weight decay and tune δ
- Batch normalization also acts as a regularizer
- Consider dropout if network still overfits

Always use early stopping and with $e > 1$

Basic design from previous lecture

- Depth of 10 (number of conv and linear layers)
- Not deep enough for optimal performance



Increasing depth is simple in theory

- Just add more conv layers between pooling layers

In practice as a rule of thumb

- Increasing the depth like this improves performance
- With batch normalization and proper regularization
- Up to a depth of around 20

Even deeper networks eventually perform worse

- Not intuitive (network could learn identity functions)
- However doing so is challenging

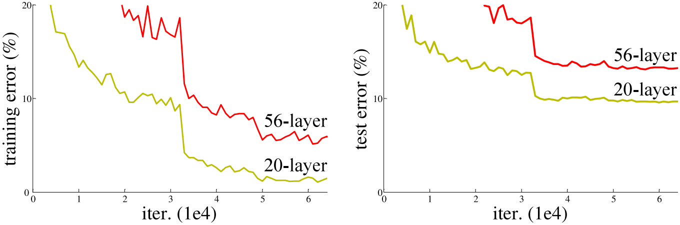


Image from [1]

Residual networks (ResNets) [1] facilitate this task

- Learn what to change about input
- Making it easy to learn identity function
- By using skip-connections and summation

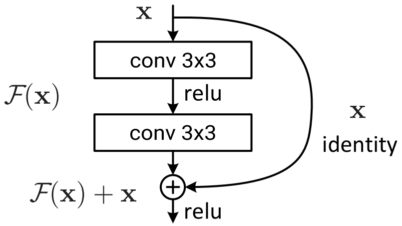


Image from [1]

Modern Classification Networks
Residual Networks

Idea is that

- Individual layers only change input a little
- But network has many more of them

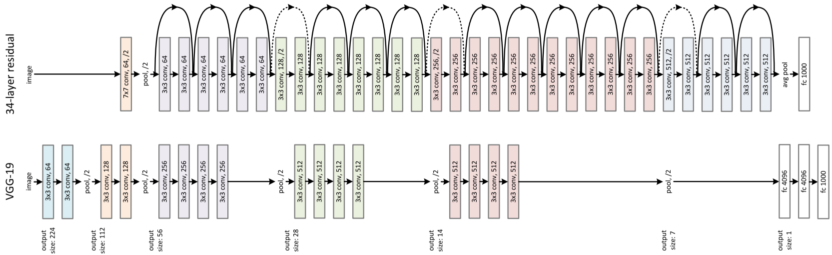


Image adapted from [1]

Another benefit is easier gradient flow

- Recall that sum nodes propagate gradients without changes
- And that gradients that join are summed

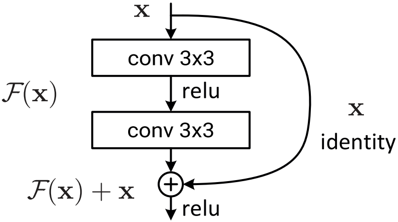
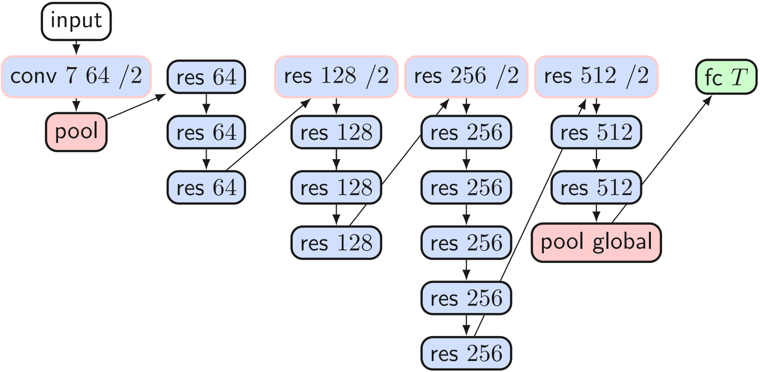


Image from [1]

ResNet-34 ($R = 224$)

- First and last layers identical to baseline design



Must support changes to C, H, W

- To enable pooling and adjusting feature map counts

1×1 (point-wise) convolutions are a flexible tool

- Can adjust C freely
- Can adjust H, W via stride

ResNet variant

- First 3×3 conv with stride 2 that doubles C
- Extra 1×1 conv with same configuration before sum

Such networks scale to depths of 1000 and more (!)

- 1202 variant overfits below (training error ≈ 0)

method			error (%)
Maxout [10]			9.38
NIN [25]			8.81
DSN [24]			8.22
	# layers	# params	
FitNet [35]	19	2.5M	8.39
Highway [42, 43]	19	2.3M	7.54 (7.72±0.16)
Highway [42, 43]	32	1.25M	8.80
ResNet	20	0.27M	8.75
ResNet	32	0.46M	7.51
ResNet	44	0.66M	7.17
ResNet	56	0.85M	6.97
ResNet	110	1.7M	**6.43** (6.61±0.16)
ResNet	1202	19.4M	7.93

Image from [1]

Very deep ResNets use more efficient blocks

- Reduce C before expensive 3×3 conv (bottleneck layer)

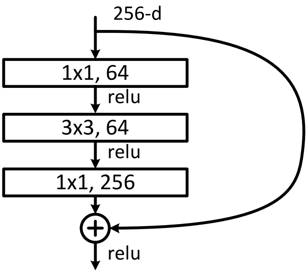


Image from [1]

ResNets still perform great today

- Accuracy typically within $\approx 5\%$ of state of the art
- Good default architecture for deep networks
- PyTorch: `net = torchvision.models.resnet34()`

Not optimized for efficiency though

Efficiency usually matters

- More efficienct networks cost less to operate
- Hardware limitations (e.g. mobile phones)

We focus on efficiency during inference (not training)

- In terms of accuracy vs. FLOPS (or predictions/second)

Point-wise and depth-wise convolutions are key ingredients

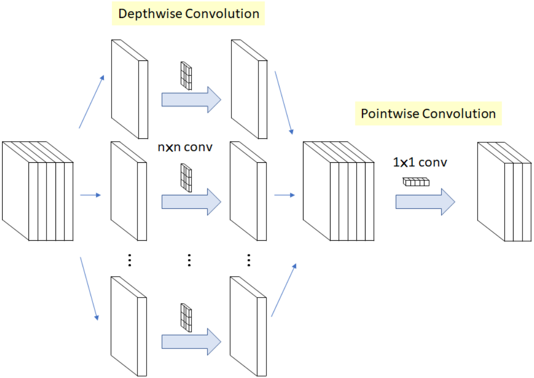


Image from mc.ai

Building blocks of MobileNet v2 [3]

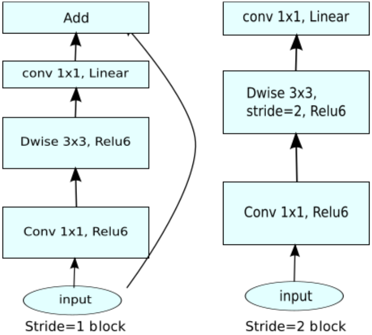
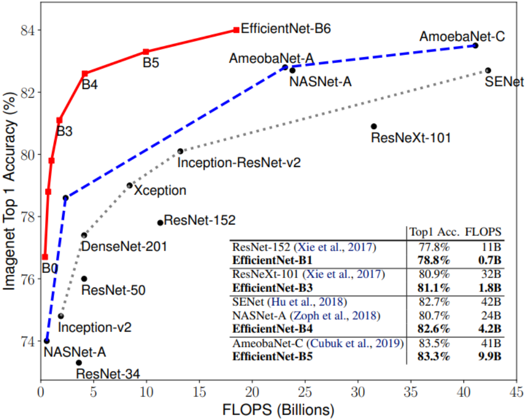


Image from [2]

Modern Classification Networks

Efficient Architectures



	Top1 Acc.	FLOPS
ResNet-152 (Xie et al., 2017)	77.8%	11B
EfficientNet-B1	**78.8%**	**0.7B**
ResNeXt-101 (Xie et al., 2017)	80.9%	32B
EfficientNet-B3	**81.1%**	**1.8B**
SENet (Hu et al., 2018)	82.7%	42B
NASNet-A (Zoph et al., 2018)	80.7%	24B
EfficientNet-B4	**82.6%**	**4.2B**
AmeobaNet-C (Cubuk et al., 2019)	83.5%	41B
EfficientNet-B5	**83.3%**	**9.9B**

Image from [3]

We previously talked about varying the depth of our networks

This is a form of network scaling

- Adapt capacity of network
- To optimize performance given task at hand
- While considering the computational budget

We will next cover other forms

Modern Classification Networks
Network Scaling

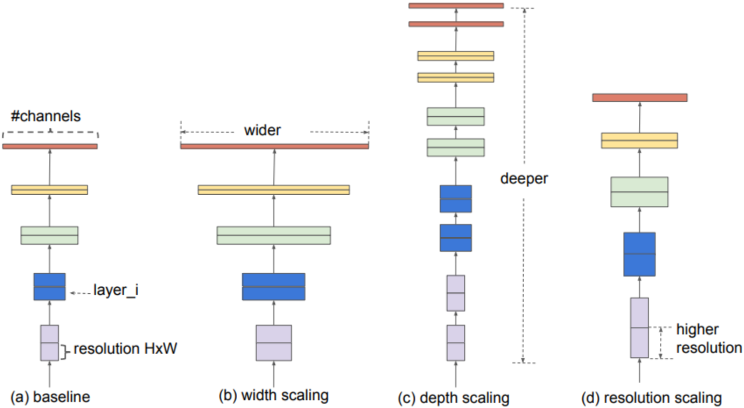
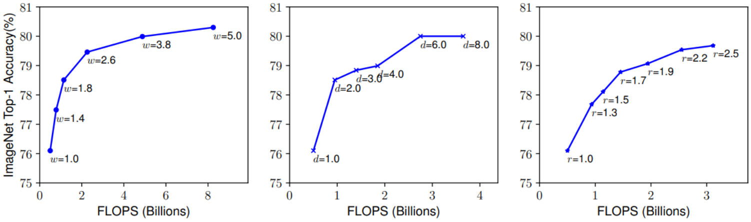


Image from [3]

Scaling width, depth, and resolution are all effective

- Impact on FLOPS varies (depends on architecture)
- Improvements compound (with diminishing returns)



Scaling by width, depth, resolution, in this order. Image from [3].

Given all this, how can we design efficient networks?

For practicioners like you

- Stick to established architectures that work well
- ResNet, EfficientNet, ConvNeXt are examples

For researchers in the field

- Manually via trial and error (based on prior work)
- Automatically via neural architecture search (NAS)

Modern Classification Networks

ConvNeXt

State of the art architecture

- Design inspired by Transformers (later)

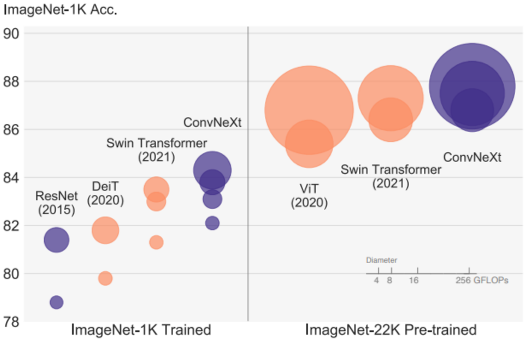


Image from [5]

Modern Classification Networks
ConvNeXt

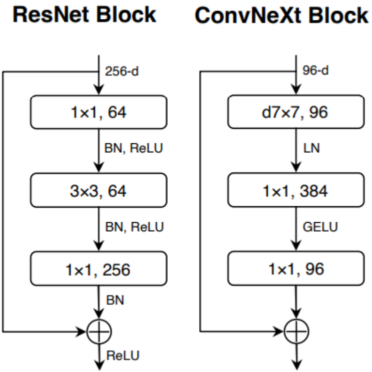


Image from [5]

Modern Classification Networks
ConvNeXt

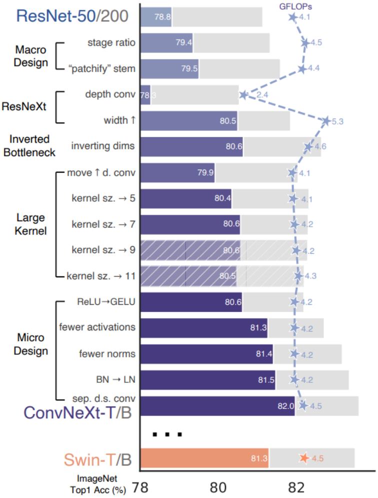


Image from [5]

Data augmentation & regularization cannot replace data

- Deep learning scales extremely well with data
- So obtaining more data should be a priority

For this reason, a powerful technique is fine-tuning

- Pre-train network on related large dataset (e.g. ImageNet)
- Then fine-tune same network on smaller target dataset

Idea is to exploit information present in both datasets

For instance we could

- Pre-train a network on ImageNet (1000 classes)
- Then replace its linear classifier with one for 2 classes
- And fine-tune the network on our cats vs. dogs dataset

Fine-tuning almost always helps

- Even if target dataset is large itself
- Assuming both datasets are related (the more, the better)

If the target dataset is very small

- Consider fine-tuning just the new classifier
- Can be done by freezing parameters of other layers

If not, a good strategy is to

- First fine-tune only new classifier
- Then unfreeze other layers and train some more
- Using a larger learning rate for classifier than for rest

Pushing Classification Performance
Handling Imbalanced Data

So far we have assumed balanced data

- Same number of samples per class

If this is not the case

- Classifier will pay more attention to majority classes
- Leading to bad accuracy for minority classes

This is usually not desired and must be addressed

- We will cover two popular options

Via class weights on the loss

- Multiply per-sample losses by weights
- With weights based on inverse class frequencies
- Easy to implement and no computational overhead

Via oversampling of the training data

- Draw samples with replacement to balance class frequencies
- Works well in combination with data augmentation
- Considerable computational overhead

Remarks

We will now move on from classification to other tasks

- Most architectures use a classification network as backbone
- Train a classification network on large dataset (e.g. ImageNet)
- Adapt network for e.g. object detection

This is a form of transfer learning

- Adapt a model trained on one task to perform well on another

Bibliography

[1] He et al. Deep Residual Learning for Image Recognition. 2015

[2] Sandler et al. MobileNetV2. 2018

[3] Tan & Le.EfficientNet. 2019

[4] Srivastava et al. Dropout. 2014

[5] Liu et al. A ConvNet for the 2020s. 2022