INTRODUCTION

IN 5400— Training convolutional networks for classification

Anne Solberg

27.02.2019

University of Oslo

Plan for today

- Activation functions
- Weight initialization
- Data normalization
- Optimization algorithms
- Learning rates
- Batch normalization
- Review well-known architectures for classification
- Hyperparameter optimization
- Overview of training
- Continued next week with topics related to generalization/regularization

Reading material and relevant video links:

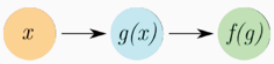
- Sections on weight initialization, data normalization, and batch normalization see http://cs231n.github.io/neural-networks-2/f
- About the learning process and optimization http://cs231n.github.io/neural-networks-3/
- Relevant video links: Lecture 6, 7 and 9 from CS 231n at Stanford, link here

The core of backpropagation

Monitor the range of inputs, weights, and outputs to make sure gradients behave well!

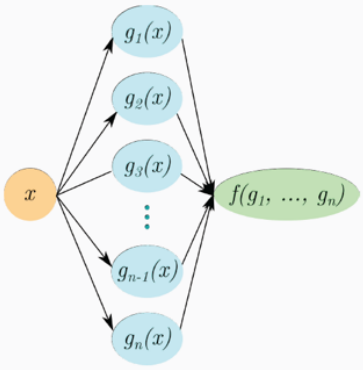
For a function f dependent on g which is dependent on x

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g}\frac{\mathrm{d}g}{\mathrm{d}x}$$



For a function f dependent on multiple $g_1, \ldots, g_n$, all which are dependent on x

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}$$



ACTIVATION FUNCTIONS

Activation functions overview

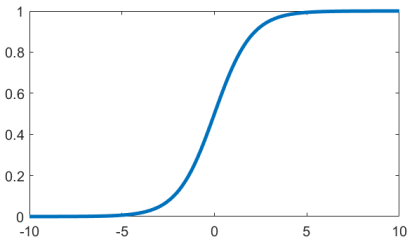
- Still an active area of research
- Sigmoid activation
- Tanh activation
- ReLU activation
- Mention briefly:
  - Leaky ReLU
  - Maxout
  - ELU
- **Goal: understand the pro's and con's of each type**

Sigmoid activation

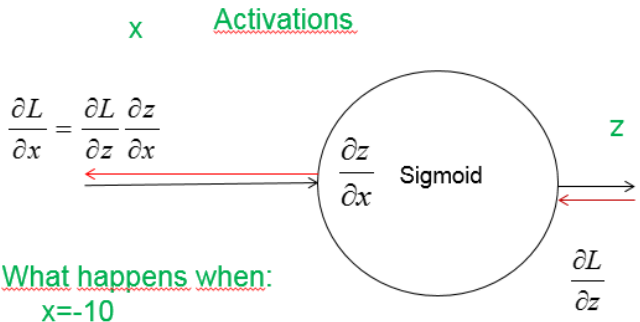
$$g(z) = \frac{1}{1 + e^{-z}}$$

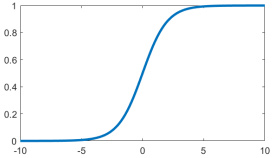
$$g\prime(z) = g(z)(1 - g(z))$$

- Output between 0 and 1
- Historically popular
- Has some shortcomings



What happens to the gradients during backpropagation?





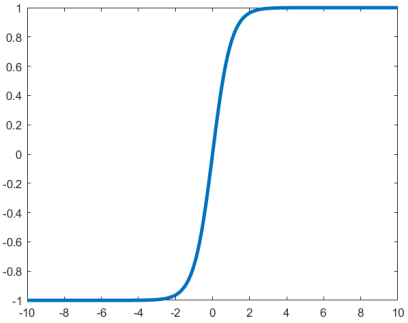
- Sigmoids kill gradients - why?
- Sigmoids are currently rarely used.

Tanh activation

$$g(z) = tanh(z)$$

$$\frac{\partial g}{\partial z} = 1 - tanh^2(z)$$

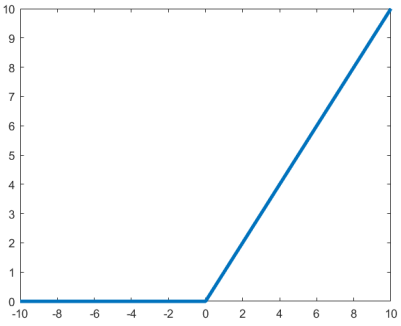
- Output between -1 and 1
- Will saturate and kill gradients (called vanishing gradients)
- Not used for convolutional layers, but used in recurrent nets where the node gets a new input for each stage (see later lecture)



Rectified Linear Unit (ReLU) activation

$$g(z) = max(0, z)$$

$$\frac{\partial g}{\partial z} = 0 \text{ if } z \leq 0, \text{ otherwise } 1$$

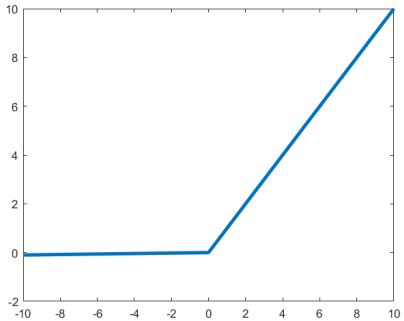


- Does not saturate
- Can sometimes 'die' during training (if input negative, output is 0)
- Can sometimes result in exploding gradients, but this can be limited by using Batch Normalization
- Be careful with the learning rate
- **Best overall recommendation for convolutional layers**

Leaky ReLU activation

$$g(z) = max(0.01z, z)$$

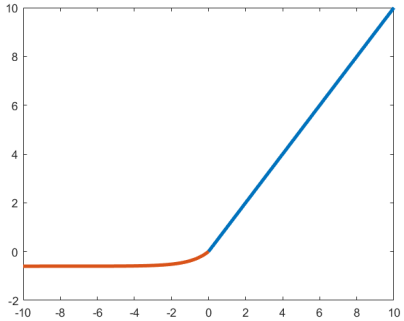
- Does not saturate
- Less prone to die
- Results are not consistently better than ReLU, and computation slighly slower
- Do the basic experiments with ReLU, but consider trying Leaky ReLU as part of the hyperparameter grid optimization
-



ELU ACTIVATION

$$g(z) = z \text{ if } z \geq 0,$$
$$\text{otherwise } \alpha(exp(z) - 1)$$

- Does not saturate or die
- Benefits of ReLU, but more expensive to compute
-
- Do the basic experiments with ReLU, but consider trying ELU as part of the hyperparameter grid optimization
-



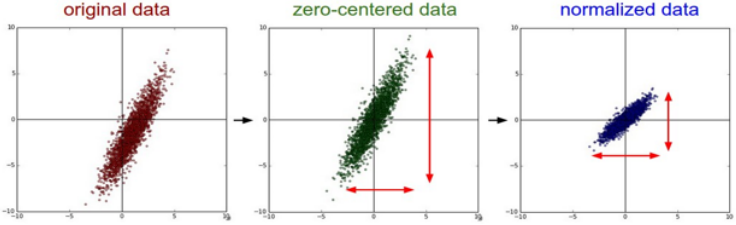
DATA NORMALIZATION

Input data scaling and gradient descent

- If the input channels or features have equal scaling, the criterion function can ideally locally look like a circle, making gradient descent behave well
- If the input channels have different scaling, the criterion function can locally look like an ellipse, making gradient descent updates overshoot and converge slow.

Common data normalization

- Most common normalization: normalize to zero mean, unit variance data by scaling each feature individually as $x = (x - \mu)/\sigma$.
- μ is the mean over the training data set, and σ the standard deviation (can be updated iteratively for each minibatch).
- For images, compute the mean and standard deviation along each channel of the data (e.g. R,G,B)
- For other types of data, consider whitening them by applying e.g. PCA if they are highly correlated
- Store the values used on the normalization, so you can apply the same normalization to the test data



WEIGTH INITIALIZATION

Weight initialization - first principle

- Do not initialize the weigths to the same value! If you do, all gradients will be equal, and all nodes will learn the same
- Break the symmetry by initializing the weights to have small random numbers.
- For deep networks, initialization should be properly scaled.
  - Too small weights, gradients die.
  - Too large weights, gradients explode.

Xavier and He initialization

- Main idea: normalize the weights in a layer to a region where gradient updates work. Normalization factor depends on the activation function.
- Consider a neuron with n inputs and $z = \sum_{i=1}^{n} w_j x_j$ (n is called the fan-in of the node).
- The variance of z is

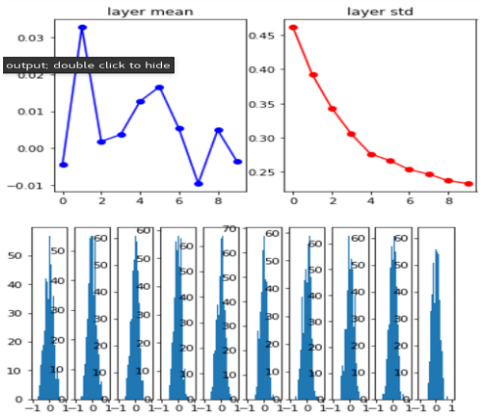
$$Var(z) = Var(\sum_{i=1}^{n} w_j x_j)$$

- It can be shown that

$$Var(z) = (n(Var(w))Var(x)$$

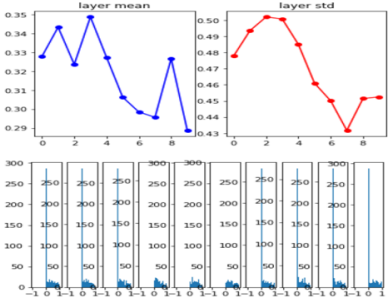
- If we make sure that $Var(w_j) = 1/n$ for all j, by scaling each weight by $\sqrt{1/n}$, the variance of the output will be 1. This works best for tanh activations.
- He showed that for ReLU, the adding a factor or $\sqrt{2}$ works better, so each weight is scaled by $\sqrt{2/n}$

Activation histogram - Xavier initalization and tanh-activation



- Network with 8 layers
- Shows histograms of the activations during training
- Activations are in a range where backpropagation works.

Activation histogram - He initalization and ReLU-activation



Initalization the bias terms

- When W is initalized to small random numbers, symmetry is broken and b can be initialized with 0.
- It is also common to initialize all b's to a common constant, e.g. 0.01

BATCH NORMALIZATION

Introduction to batch normalization

- As training progresses, the and variance of the weights will typically change, and at a certain point make convergence slow. This is called an internal covariance shift.
- Batch normalization https://arxiv.org/abs/1502.03167 countereffects this.
- Batch normalization layer are typically inserted after fully connected (or convolutional) layers, before nonlinearity.
- The normalization makes the result ($z - values$) gaussian with zero mean and unit variance.

Batch normalization algorithm - training

- For a given node and a given minibatch, compute the mean μ_k and σ_k.
- First, create zero mean, unit variance: $\hat{z}_k = (z_k - \mu_k)/\sigma_k$
- Experiments have shown that we should allow a scaling to avoid limiting what the node can express:

$$\tilde{z_k} = \gamma \hat{z}_k + \beta$$

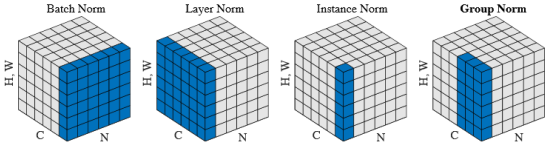
- γ andβ are learned using backpropagation, and they are specific to the layer.
- Using this scaling normally speeds up the convergence by getting more effective gradients.
- **Batch normalization significantly speeds up gradient descent and even improves performance, try it!**
- Store γ and β.

Batch normalization - what to do at test time

- At test time: we need mean and standard deviation should we use to normalize.
- The best is to use the mean and standard deviation over the entire training data set.
- This can be efficiently computed using moving average estimates over the mini batches, apply this during traning and store μ_k andσ_k.

Other types of normalization

- Normalization is important for efficient gradient flow
- Batch norm normalize across a batch, but we can also normalize across channels, instances, or groups of channels.

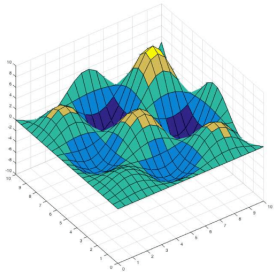


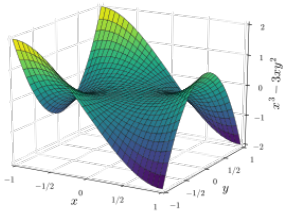
- A good reference is https://arxiv.org/pdf/1803.08494.pdf

GRADIENT DESCENT UPDATES

Problems with stochastic/minibatch gradient descent

- In low dimensions, local minima are common
- In high dimensions, saddle points are more common.





Gradient descent with momentum

- A few variations of weight updates in minibatch gradient descent are worth considering.
- Regular gradient descent updates:

$$w = w - \lambda \partial w$$

- **Momentum updates:**

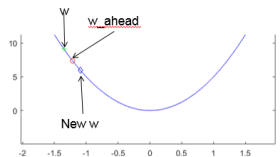
$$v = \rho v - \lambda \partial w$$
$$w = w + v$$

- ρ is the momentum parameter (common value 0.9)
- Analogy to a ball sliding down a hill. At a given point the ball has velocity v. ∂w is the local slope, and the step is determined by a combination of slope and velocity. ρ acts like friction.
- Initialize with $v = 0$.

Gradient descent with Nesterov momentum

- Idea: if we are at point w, with momentum the next estimate is $w + \rho * v$ due to velocity from previous iterations.
- Momentum update has two parts. One due to velocity, and one due to current gradient.
- Since velocity is pushing us to $w + \rho * v$, why not compute the gradient at point $w + \rho * v$, not point w? (Look ahead)
- **Nesterov momentum updates:**

$$w_{ahead} = w + \mu v$$
$$v = \mu v - \lambda \partial w_{ahead}$$
$$w = w + v$$



Implementing Nesterov

- Notice that Nesterov creates the gradient at w_{ahead}, while we go directly from w to $w + v$.
- It is more convenient to avoid computing the gradient at a different location by rewriting as:

$$v_{prev} = v$$
$$v = \rho v - \lambda \partial w_{ahead}$$
$$w = w + (1 - \rho)v - \rho v_{prev}$$

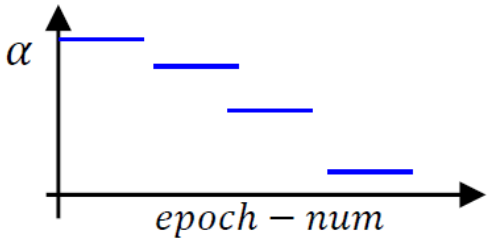
Learning rate decay

- Setting the learning rate is difficult, convergence is sensitive to it.
- Using a decay scheme is often effective:

$$\lambda = 0.95^{\text{epochnum}} \lambda_0$$

$$\lambda = \frac{k}{\sqrt{\text{epochnum}}} \lambda_0$$

- This typically helps to get away from saddle point and the loss function often looks like a staircase:



Parameter-specific update schemes

- SGD and SGD with momentum updates all weight/parameters using the same scheme
- Other methods scale the update by the size of the weights/parameters:
- We will look at ADAM, but other choices are AdaGrad or RMSprop.
  - AdaGrad http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
    - Accumulates weight gradients, these can build up and is not so often used.
  - RMSprop
    - Introduce a cache of moving average of the gradients of each weight
  - ADAM https://arxiv.org/abs/1412.6980
    - Combines both momentum and a moving average of the gradients of each weight

ADAM - MAIN IDEA WITHOUT FIX FOR THE FIRST ITERATIONS

ADAM update, all variables are vectors

1. Set $\rho_1 = 0.9$, $\rho_2 = 0.999$, $\epsilon = 1e-8$.
2. Update the mean (first order moment) $\mu_{\partial w}$ and the non-centered variance (second order moment) $var_{\partial w}$ of ∂w.

$$\mu_{\partial w} = \rho_1 \mu_{\partial w} + (1 - \rho_1)\partial w$$
$$var_{\partial w} = \rho_2 var_{\partial w} + (1 - \rho_2)(\partial w)^2$$

3. Take a scaled step:

$$w = w - \lambda \frac{\mu_{\partial w}}{(\sqrt{var_{\partial w}} + \epsilon)}$$

ADAM - ALGORITHM

ADAM update with bias correction for the first iterations:

- Set $\rho_1 = 0.9$, $\rho_2 = 0.999$, $\epsilon = 1e-8$.
- For $t = 1 : maxiter$

$$
\begin{aligned}
l\mu_{\partial w} &= \rho_1 \mu_{\partial w} + (1-\rho_1)\partial w \\
\mu_t &= \mu_{\partial w}/(1-\rho_1^t) \\
var_{\partial w} &= \rho_2 var_{\partial w} + (1-\rho_2)(\partial w)^2 \\
v_t &= var_{\partial w}/(1-\rho_2^t) \\
w &= w - \lambda \frac{\mu_t}{(\sqrt{v_t}+\epsilon)}
\end{aligned}
$$

Recommendations for weigth updates

- SGD with momentum, try learning rate decay too.
  - Also used weigth decay - covered in next lecture
- ADAM also works well
  - If using weigth decay, be aware that this a currently a matter of discussion https://www.fast.ai/2018/07/02/adam-weight-decay/

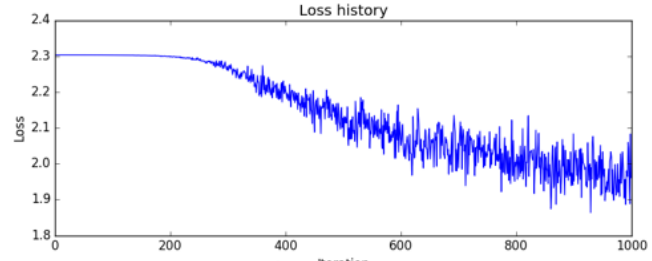
Optimizing hyperparameters

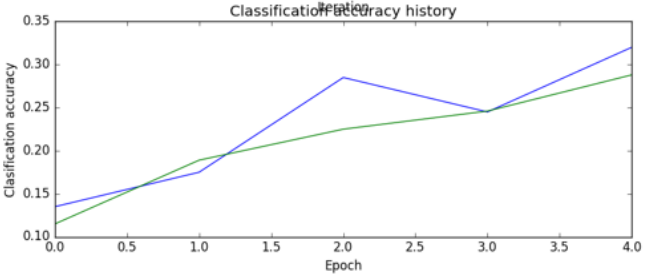
- **Training data:** part of the data set used to estimate the weights during backpropagation. For large networks, 80% of the data can be used for training.
- **Validation data:** part of the data used for finding hyperparameters (typically 10%).
- **Test data:** part of the data used for estimating the accuracy. Used ONCE after fitting all parameters and architectures.
- If you split your data set yourself, take care to avoid domain shift. Also take care not to be tempted to try the test set too early, finish all training before using it.

Training strategies

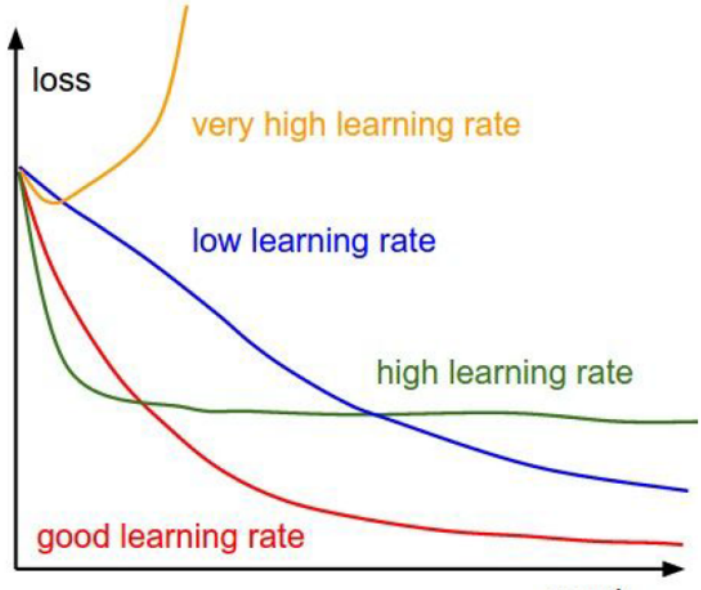
- Check if you model work on the training data set. A good idea can be to verify that you can overfit to a smaller portion of the training data. If not, your model is not working, you might have an error somewhere, or your data might not be trainable.
- Run a few epochs (iterations through all traning samples) using varying learning rate and inspect plots of the loss function and the accuracy.
- Check your amount of data compared to the model complexity: should you consider traning from scratch or using pretrained model/transfer learning (next week).
- Check preprocessing/data standardization.
- Check the quality of your labels (sometimes needed for new data sets).
- Hyperparameter search is something you start after a good set of experiments, when you have confidence that your model is working.
- Watch out for NaN! Error? Too high learning rate?
- Use regularization/dropout (learn about this next week)!
- Experience helps!

Monitor the loss and accuracy for each run





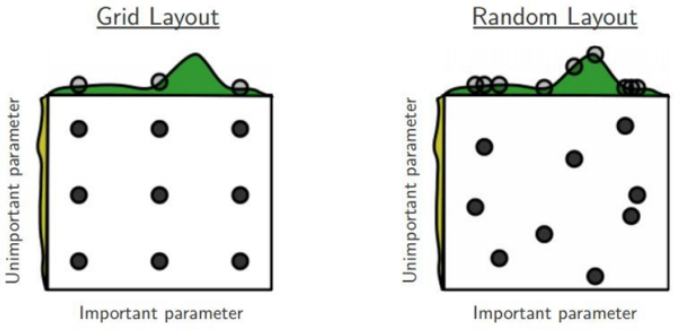
Loss functions and learning rate



Hyperparameters: network architecture

- Number of layers
- Number of filters in each layer
- Filter kernel size

Grid search



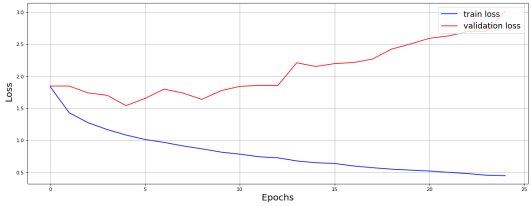
- Use a random grid!

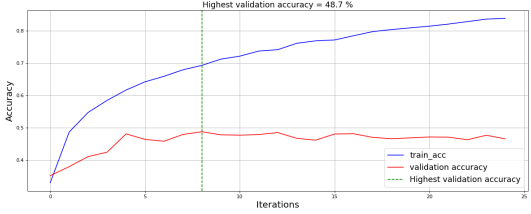
Scale for hyperparameters

Parameter	Initial Value	Scale
Learning rate	0.001 or 0.01	Log
Momentum	0.9	Log
Mini-batch size	32, 64, 128	Linear
Number of layers		Linear
Number of hidden units		Linear
Filter kernel size	3×3	Linear

- Hyperparameters are not orthogonal
- They can have different sensitivity
- Next week: add Dropout, weight decay and other regularization parameters to the parameter list

Typical results after traning using these methods





- Gap between training error and validation error .
- Need regularization to avoid overfitting (next lecture).

CNN ARCHITECTURES FOR CLASSIFICATION

Why look at previous architectures?

- Understand how the basic parts of a successful CNN work together
- Understand how the shape of the output changes
- Understand the new ideas introduced in central architectures
- These architectures are the starting point for your training on new classification applications.
- Many works refer to these.
- Remembering details not essential!

LeNet - Short introduction

Paper	Gradient Based Learning Applied to Document Recognition
Authors	Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner
Year	1998
Citations	11 135
Link to	pdf

Initial notes

- Very influential, and successful in its time
- First “modern” cnn
- We start to see tendencies of the familiar cnn composition, but it is not the first cnn
- The paper discusses a lot of central aspects
- Also uses a lot of deprecated techniques:
  - Originally uses a *stochastic diagonal Levenberg-Marquardt* optimization routine
  - Originally uses distance from an “ideal” set of ASCII characters as loss
  - The “idea” of the method holds with SGD and softmax
  - Originally a complicated scheme of which filters to apply on which feature maps
  - Also uses non-linearity after pooling

Activation functions

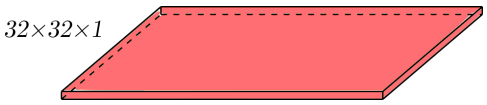
- Convolution nodes uses a scaled $\tanh$ non-linearity

$$g(z) = A \tanh(Sz) \tag{1}$$

- Sets $A = 1.7259$, and $S = 2/3$
- This makes $g(-1) = -1$ and $g(1) = 1$, which is chosen for convenience

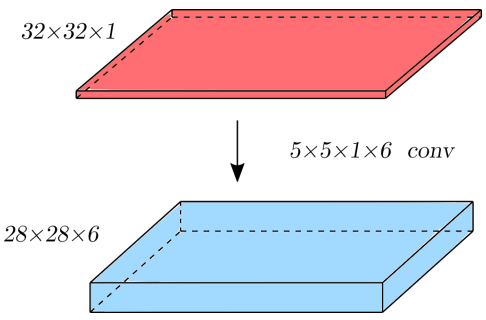
LeNet — Input

- $32 \times 32 \times 1$
- Used for character recognition
- Normalized to zero mean and unit variance



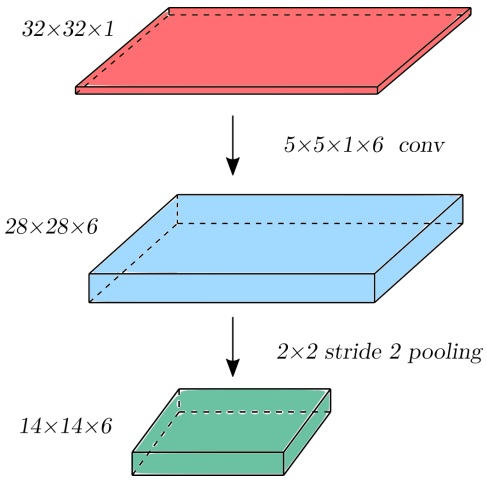
LeNet — First convolutional layer

- Input shape: $32 \times 32 \times 1$
- 6 convolutions with kernel shape $5 \times 5 \times 1$, no padding
- $5 \cdot 5 \cdot 6 + 1 \cdot 6 = 156$ trainable parameters
- Output shape: $28 \times 28 \times 6$



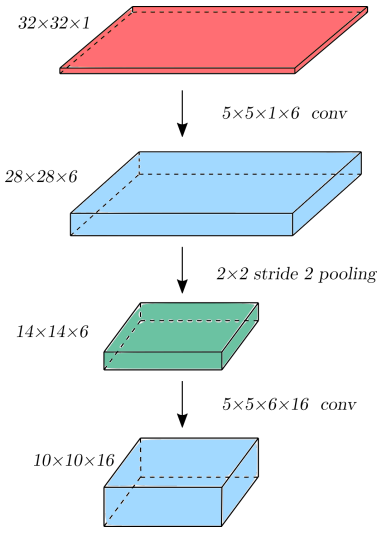
LeNet — First subsample (pooling) layer

- Input shape: $28 \times 28 \times 6$
- Window shape: 2×2 with stride 2
- Output shape: $14 \times 14 \times 6$
- Activation for a unit: $a = g\left(\frac{x_1+x_2+x_3+x_4}{w} + b\right)$
- w and b is shared by all units in a feature map
- w and b are trainable, resulting in $6 \cdot (1+1) = 12$ parameters
- Very similar to an average pool layer



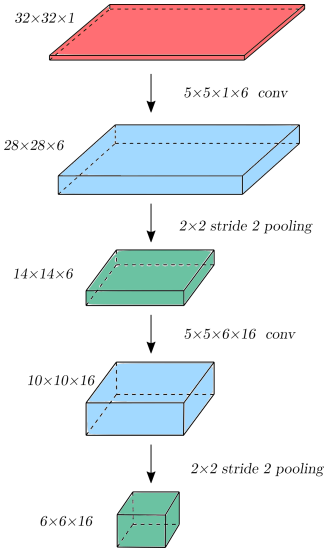
LeNet — Second convolutional layer

- Input shape: $14 \times 14 \times 6$
- 16 convolutions with shape $5 \times 5 \times 6$, no padding
- Output shape: $10 \times 10 \times 16$
- In total
  $25 \cdot (6 \cdot 3 + 6 \cdot 4 + 3 \cdot 4 + 6) + 16 = 1516$
  trainable parameters



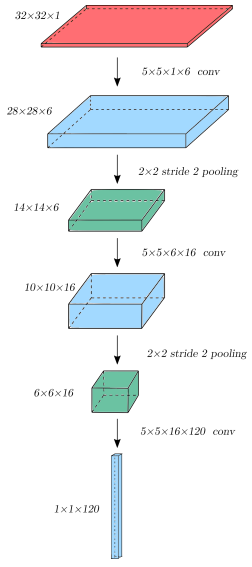
LeNet — Second subsample (pooling) layer

- Input shape: $10 \times 10 \times 16$
- Window shape: 2×2 with stride 2
- Output shape: $5 \times 5 \times 16$
- Activation for a unit: $a = g\left(\frac{x_1+x_2+x_3+x_4}{w} + b\right)$
- w and b is shared by all units in a feature map
- w and b are trainable, resulting in $16 \cdot (1+1) = 32$ parameters



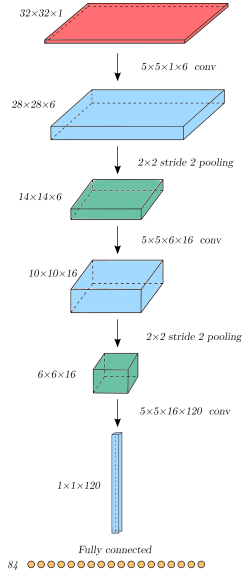
LeNet — Third convolutional layer

- Input shape: $5 \times 5 \times 16$
- 120 convolutions with shape $5 \times 5 \times 16$, no padding
- Output shape: $1 \times 1 \times 120$
- In total $5 \cdot 5 \cdot 16 \cdot 120 + 1 \cdot 120 = 48120$ trainable parameters



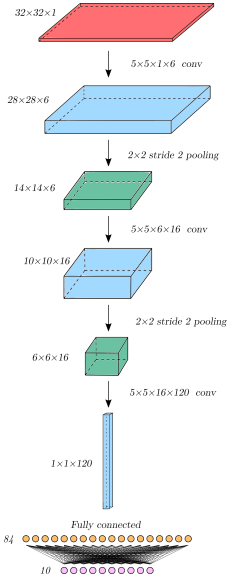
LeNet — Fully connected layer

- Input nodes: 120
- Output nodes: 84
- In total $120 \cdot 84 + 84 = 10164$ parameters



LeNet — Fully connected output layer

- Input nodes: 84
- Output nodes: 10 (number of classes in MNIST)
- In total $84 \cdot 10 + 10 = 850$ parameters



Architecture — summary

- Alternates between convolution and pooling layers, finishing with dense layers
- Propagating through the network:
  - Number of channels (feature maps) increase
  - Feature map dimensions reduce
- Number of trainable parameters: 60 850

AlexNet - Short introduction

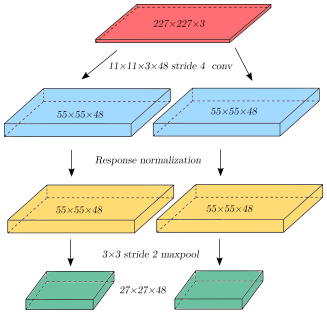
Paper	ImageNet Classification with Deep Convolutional Neural Networks
Authors	Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton
Year	2012
Citations	20 340
Link to	pdf

Initial notes

- At the time superior performance on the ImageNet challenge
- Kick-started the machine-learning renaissance
- Hinted at the importance of depth
- Successful use of dropout and ReLU
- Very efficient convolution implementation
- Distributed the network over 2 GPU's

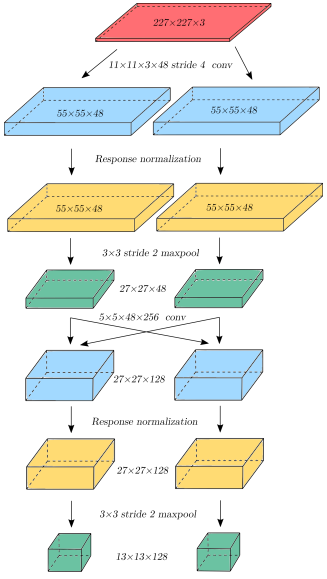
Architecture — First convolution

- Input shape: $227 \times 227 \times 3$
- On each gpu: 48 $11 \times 11 \times 3$ convolutions with stride 4
- Response normalization
- 3×3 max pool with stride 2
- Output shape: $27 \times 27 \times 48$ on each gpu



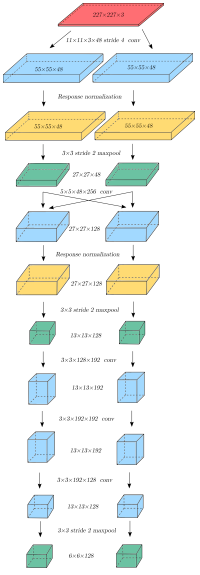
Architecture — Second convolution

- Input shape: $27 \times 27 \times 48$
- On each gpu: $128\ 5 \times 5 \times 256$ convolutions
- Notice the communication between gpus
- Response normalization
- 3×3 max pool with stride 2
- Output shape: $13 \times 13 \times 128$ on each gpu



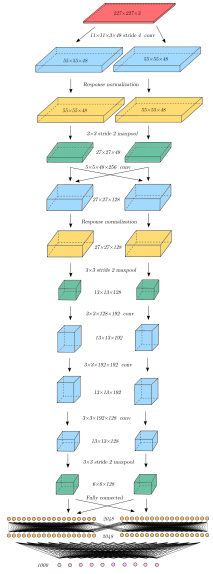
Architecture — last convolution layers

- On each gpu:
  - Input shape: $13 \times 13 \times 128$
  - Conv3: 192 $3 \times 3 \times 128$ convolutions
  - Conv4: 192 $3 \times 3 \times 192$ convolutions
  - Conv5: 128 $3 \times 3 \times 192$ convolutions
- 3×3 max pool with stride 2
- Output shape: $6 \times 6 \times 128$ on each gpu



Architecture — dense output layers

- On each gpu:
  - Input shape: $6 \times 6 \times 128$
  - Dense1: $9216 (= 2 \cdot 6 \cdot 6 \cdot 128) \rightarrow 4096$
  - Dense2: $2048 \rightarrow 4096$
  - Dense3: $2048 \rightarrow 1000$
- Notice communication between gpus
- Final output (1000) is the number of classes



Summary

- Alternating convolution and pooling, finalizing with dense layers
- Reducing spatial dimension, and increasing number of feature maps
- Uses ReLU
- Uses data augmentation, weight decay, and dropout
- Very many parameters compared to LeNet, about 60 million

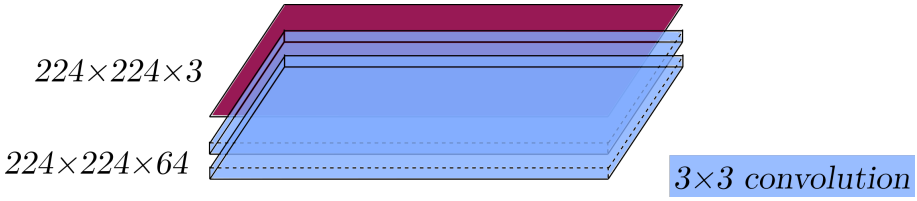
VGG - Short introduction

Paper	Very Deep Convolutional Networks for Large-Scale Image Recognition
Authors	Karen Simonyan, and Andrew Zisserman
Year	2014
Citations	9428
Link to	pdf

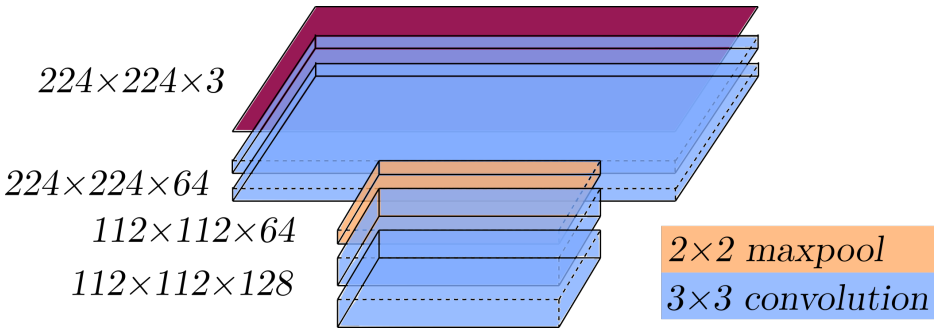
Initial notes

- Simple and elegant design
- Further investigates the importance of deep nets
- Very good performance on ImageNet
- Very large

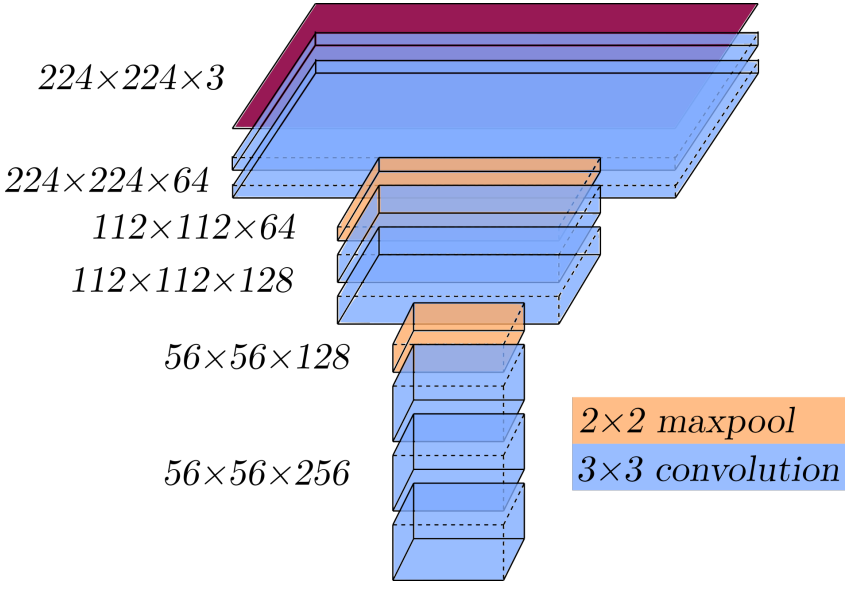
VGG16 — First convolution



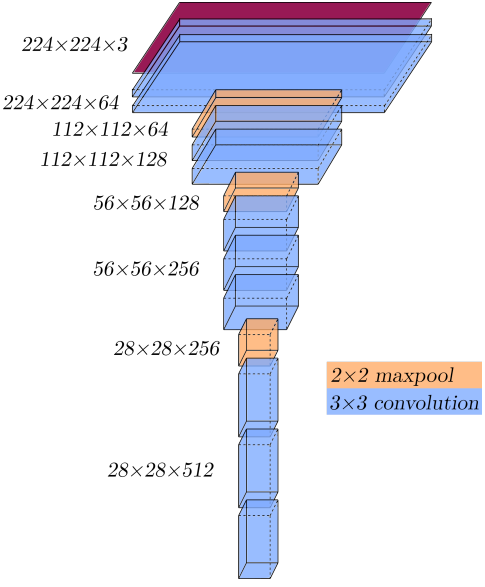
VGG16 — First downsampling



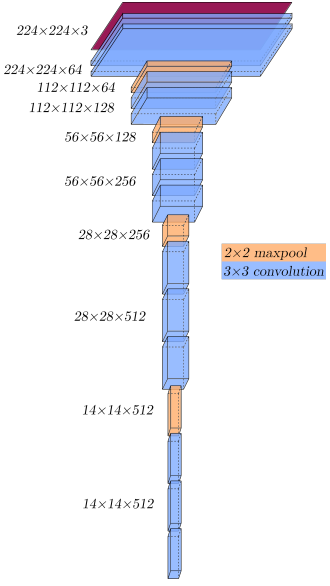
VGG16 — Second downsampling



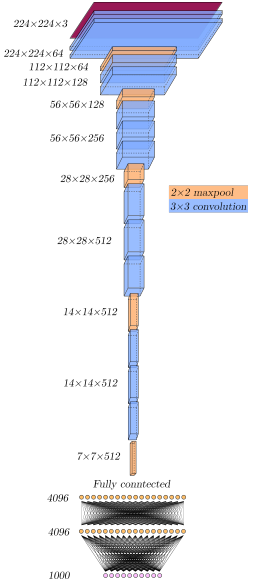
VGG16 — Third downsampling



VGG16 — Fourth downsampling



VGG16 — Output layers



Paper	Going deeper with convolutions
Authors	Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich
Year	2014
Citations	6282
Link to	pdf

Initial notes

- Impressive ImageNet result
- Complex structure with few parameters (anti-thesis of VGG networks)
- 12 times fewer parameters than AlexNet

GoogLeNet — Stacking inception modules

- The inception module controls the number of feature maps
- Can stack multiple inception modules
- Put max-pool layers in between occasionally

ResNet - Short introduction

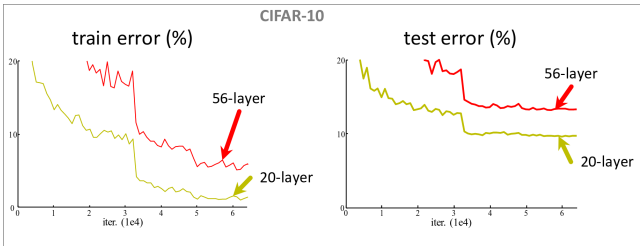
Paper	Deep Residual Learning for Image Recognition
Authors	Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun
Year	2015
Citations	6598
Link to	pdf

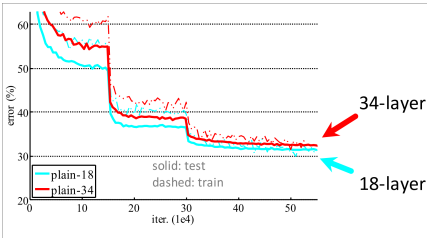
Initial notes

- “Solved” ImageNet
- Elegant solution to a concrete problem

Problem

- Deeper models seems to be better
- However, very deep models perform worse
- Not due to overtraining
- Degradation problem



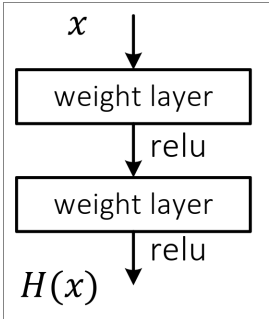


Deep models should be at least as good as shallow

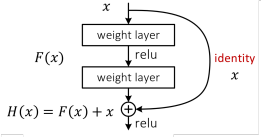
- A deeper model should not have higher training error
- “Proof” by construction
  - Take a shallow model
  - Insert extra layers as identity mappings
  - This deeper mode should have at least as good training error
- How to solve this is the key

Residual learning

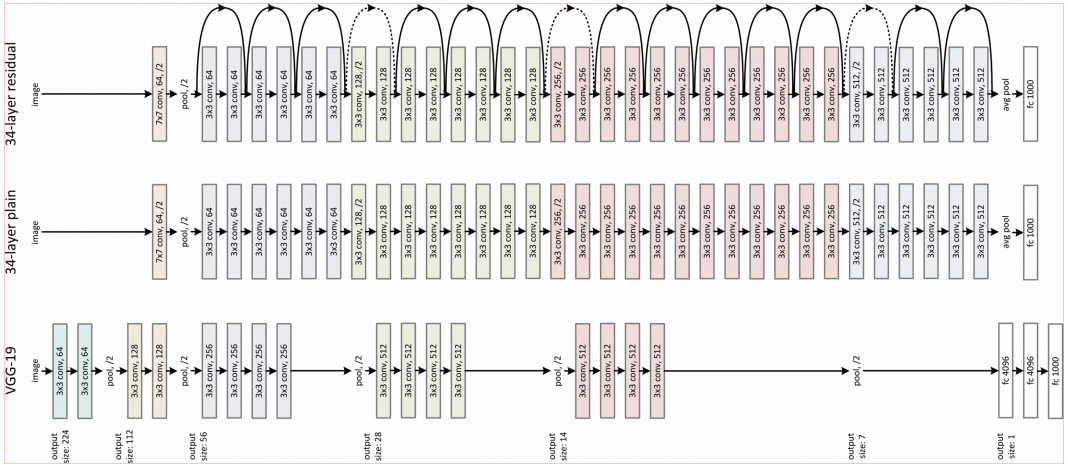
- Stack a couple of layers
- Input x
- Let $H(x)$ be the desired mapping to be learned



- Explicitly compose the output as $H(x) = F(x) + x$, by adding the input x
- This means that what has been learnt is the residual $F(x) = H(x) - x$
- This should make identities $H(x) = x$ easier to learn
- Easier to train very deep networks



Base architecture



Revolution of depth

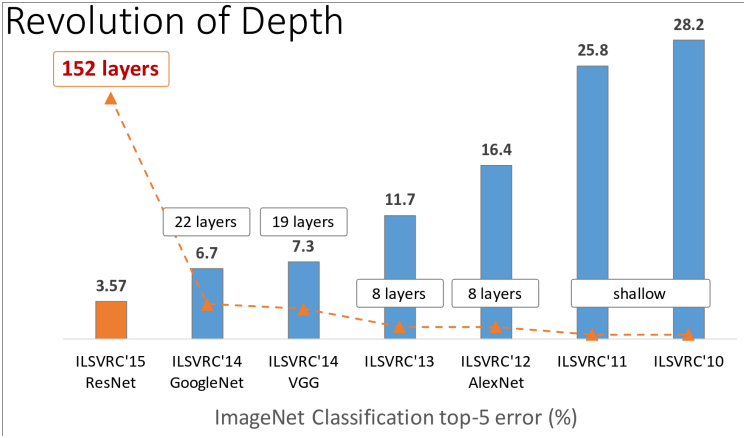


Figure 1: Source: *An analysis of Deep Neural Network Models for Practical Applications.* Canziani, A., Paszke, A., Culurciello, E., 2016

ImageNet accuracy and size

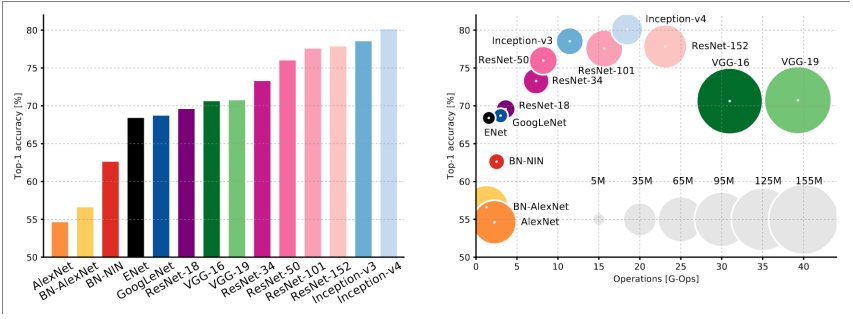


Figure 2: Size and accuracy comparison. Blob size reflects the number of parameters. Source: *An analysis of Deep Neural Network Models for Practical Applications*. Canziani, A., Paszke, A., Culurciello, E., 2016

Thoughts about architecture

- As with everything: choose the tool best suited for your problem
- ImageNet top accuracy is not necessarily your ideal metric
- Some things (non-exhaustive) to take into account, in addition to accuracy
  - Training time
  - Inference time
  - Power consumption
  - Memory consumption
  - Processing power consumption
  - Amount of training data
- Hard constraints on the above have shaped current models

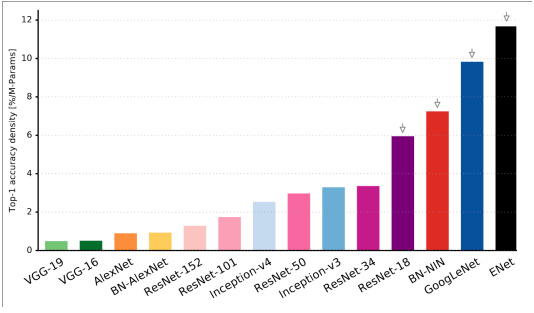


Figure 3: Source: *An analysis of Deep Neural Network Models for Practical Applications.* Canziani, A., Paszke, A., Culurciello, E., 2016

Learning goals

- Pro's and con's for differnet activation functions.
- How weights should be initialized and scaled given the activation function.
- How batch norm works at traning and test time.
- How momentum SGD and ADAM works.
- Know how to optimize the hyperparameters, including scale and sensitivity.
- Know the most characteristic features of central architectures.

Questions?