Neural networks & deep learning
Neural networks, Convolutional neural networks
Lecture block 5

Anders Eklund

anders.eklund@liu.se

Department of Biomedical Engineering (IMT)
Department of Computer and Information Science (IDA)
Center for Medical Image Science and Visualization (CMIV)
Linköping University, Sweden

December 2, 2019

Outline - part 1 of 6

- Practical info
- Neural networks
  - activation functions
  - loss functions
  - regularization
  - optimization
- Back propagation

Neural networks & deep learning

- Part 1: Neural networks, back propagation
- Part 2: Basics of deep learning
- Part 3: Convolutional neural networks (classification, segmentation)
- Part 4: Image synthesis using generative adversarial networks
- Part 5: Recurrent neural networks
- Part 6: Practical methodology (DL frameworks, Docker, GPUs, ...)

My background

- Y-programme (applied physics & electrical engineering) 2007, profile signal & image processing
- PhD 2012, medical image processing
- Doing research about image processing, machine learning and statistics

Book

- https://www.deeplearningbook.org
- Available for free online, but not as pdf
-

- Chapters 6, 7, 8, 9, 10, 11, 20.10.4

Laborations

- Computer laborations will be done in Jupyter notebooks
- Programming using Keras with Tensorflow backend
- Run laborations through the Microsoft Azure cloud,
  you get 10 hours with an Nvidia graphics card
- You connect to the cloud notebook using an ssh tunnel,
  run Jupyter notebook through your web browser
- You need ssh, exists in Mac and Linux,
  download "Putty" for Windows

GOOGLE COLAB

- If you need more GPU time, import your Jupyter notebook to Google Colab, colab.research.google.com
- Free GPU acceleration (restrictions ?)

-

Laborations

- Part 1: Image classification using 2D CNNs
  - Classify an image into one of 10 classes
- Part 2: Image segmentation using 2D CNNs
  - Segment blood vessels from a retinal image

Neural networks

- Basic building block of deep learning
- Takes an input vector x of some dimension, outputs label or number
- Every layer of the neural network applies a weight vector (scalar product) and a (non-linear) activation function
  - Feedforward network, information flows from input layer to output layer (no feedback connections)
  - Recurrent network, also includes feedback connections (part 5)

Neural networks (feedforward)

Input x, number of nodes = number of dimensions in data
Hidden layers, new features, one weight vector per node per layer
Output y, number of nodes = number of classes, strong. activation = class

Neural networks - components

- Activation function for each layer
- Loss function to minimize (regression, classification)
- Regularizer, to prevent overfitting
- Optimization algorithm

Neural networks - basics

- Training / input vectors x^μ
- Ground truth t^μ (target, e.g. 0 and 1 for two classes)
- Weights for single node w, bias b
- Output from a single node (for one training vector), $s^\mu = w^T x^\mu + b$
- Output after activation function f (prediction), $y^\mu = f(s^\mu)$
- Train network so that difference between y and t is as small as possible

Neural networks - basics

- Output from layer L can be calculated using matrix-matrix multiplication
- Put all weight vectors in a layer (one per node) in weight matrix W_L
- Put all biases (one per node) into a bias vector B_L
- Put input vectors (one per training example in batch) into matrix X
- Output from layer is given by $S = W_L X + B_L, Y = f(S)$
  (B_L is repeated to a matrix)
- S is a matrix with outputs from all nodes and all examples in batch

Neural networks - activation functions

- Need non-linear activation functions, otherwise no point of having several layers
- Sigmoid (logistic), range 0 to 1,

  $y = \frac{1}{1+exp(-s)}$
- Hyperbolic tangent (tanh), range -1 to 1,

  $y = tanh(s) = \frac{exp(s)-exp(-s)}{exp(s)+exp(-s)}$
- Softmax for k classes (outputs can be interpreted as probabilities),

  $y(i) = \frac{exp(s(i))}{\sum_k exp(s(k))}$
- See https://keras.io/activations/ for more activation functions

Neural networks - one layer result

2 linearly separable classes, one layer is enough
weight vector defines decision boundary
bias is shift from origin

Neural networks - one layer result

3 not linearly separable classes, one linear layer is not enough

Neural networks - two layers result

3 not linearly separable classes, two layers are enough

Neural networks - loss functions

- Error between ground truth (targets) t and predictions y, N examples
- Mean squared error (MSE), regression
  $L = \sum_{\mu=1}^{N} (t^{\mu} - y^{\mu})^2$
- Mean absolute error (MAE),
  $L = \sum_{\mu=1}^{N} |t^{\mu} - y^{\mu}|$
- Binary cross entropy (two classes), classification
  $L = \sum_{\mu=1}^{N} -(t^{\mu} log y^{\mu} + (1 - t^{\mu}) log(1 - y^{\mu}))$
- Categorical cross entropy (k classes), classification
  $L = \sum_{\mu=1}^{N} - \sum_{k} [t = k] log y_k$
  $[t = k]$ gives 1 if $t = k$, 0 otherwise
- See https://keras.io/losses/ for more loss functions

Neural networks - k classes

- For more than 2 classes, need to store targets as k-dimensional vectors
- $k = 5$
- Training example 1, class 1, [1, 0, 0, 0, 0]
- Training example 2, class 5, [0, 0, 0, 0, 1]
- In Keras,
  categorical_labels = to_categorical(int_labels, num_classes = 5)

Regularization techniques

- The performance is often high on the training set,
  but lower on validation and testing set
- Different regularization techniques are available
  to prevent overfitting to the training data
- Find the best weights, subject to different penalties
- Prefer networks with few and small weights
- Start with large complex network,
  using regularization will find the "best" sub-network

Regularization techniques - L1 & L2

- L1 norm (LASSO),

  $\lambda_1 \sum_i |w_i|$

- L2 norm (ridge regression / Tikhonov),

  $\lambda_2 \sum_i w_i^2$

- L1 + L2 norm (elastic net),

  $\lambda_1 \sum_i |w_i| + \lambda_2 \sum_i w_i^2$

- Add penalty term to loss function, e.g.

  $L = \sum_{\mu=1}^{N} -(t^\mu log y^\mu + (1 - t^\mu) log(1 - y^\mu)) + \lambda_2 \sum_i w_i^2$

Regularization techniques - dropout

- Dropout - randomly set hidden units to 0
  in a fully connected (dense) layer during training,
  with a given probability (e.g. 50%)
- Network cannot rely on a few connections,
  makes it better at generalizing to new data
- A way to combine many different neural network architectures
- Dropout is normally only used for training, not for testing,
  but we revisit this point in part 6

Regularization techniques - dropout

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

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

Regularization techniques - indirect

- Data augmentation
  - Increase the amount of realistic training data, by modifications of the existing training data
- Early stopping
  - Stop training when validation error increases
- Bagging / Ensemble
  - Train several models separately, let all models vote (time consuming)
- Multi-task learning
  - Train a feature extractor to be good at several tasks (e.g. classification + segmentation)

Neural networks - optimization

- Gradient descent
- Step 0: Start with some weights W
- Step 1: Compute gradient of loss function, with respect to weights, $\nabla L(W)$
- Step 2: Move a step in the negative gradient direction,

  $$W = W - \alpha \nabla L(W)$$

- α is the learning rate
- Iterate Step 1 and 2 until some stopping criteria is met

Neural networks - gradients

- How do we get the gradients for all layers, at the same time?
- Chain rule of derivatives,
  how does the loss change with respect to the weights in each layer?
- Forward propagation, feed data through network, calculate errors
- Backpropagation, go backwards, update weights

Neural networks - gradients

For mean squared error, one layer network

$$\frac{\partial L(W)}{\partial w_i} = \sum_{\mu=1}^{N} \frac{\partial L}{\partial y^\mu} \frac{\partial y^\mu}{\partial s^\mu} \frac{\partial s^\mu}{\partial w_i} = \sum_{\mu=1}^{N} 2(y^\mu - t^\mu) f'\left(s^\mu\right) x_i^\mu \tag{1}$$

Collect derivatives in vector

$$\nabla L(W) = 2 \sum_{\mu=1}^{N} (y^\mu - t^\mu) f'\left(s^\mu\right) x^\mu \tag{2}$$

Matrix form for all training examples (remove factor 2)

$$\nabla L(W) = ((y - t) .* f'\left(s\right)) X^T \tag{3}$$

Two layer network, same principle but more complex formulas...

Neural networks, 2 classes - in Keras

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation
import numpy as np

Xtrain = np.random.random((1000,20))
Ytrain = np.random.randint(2, size=(1000,1))

model = Sequential()

# First layer, 32 hidden nodes
model.add(Dense(32, activation='sigmoid',
                input_dim=20))

# Second layer, 1 output node
model.add(Dense(1, activation='sigmoid'))
```

Neural networks, 10 classes - in Keras

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation
import numpy as np

Xtrain = np.random.random((1000,20))
Ytrain = np.random.randint(10, size=(1000,1))
Ytrain = keras.utils.to_categorical(Ytrain,10)

model = Sequential()

# First layer, 32 hidden nodes
model.add(Dense(32, activation='sigmoid',
                input_dim=20))

# Second layer, 10 output nodes
model.add(Dense(10, activation='softmax'))
```

Training the neural network - Keras

```
from keras.models import Sequential, Model
from keras.losses import binary_crossentropy as BC
from keras.optimizers import SGD

model = Sequential()
...
model.compile(loss=BC, optimizer=SGD(lr=0.05),
              metrics=['accuracy'])

model.fit(Xtrain, Ytrain, batch_size=100,
          epochs=100, verbose=2,
          validation_data=(Xval, Yval))
```

See https://keras.io/models/model/#compile and https://keras.io/models/model/#fit

Using the neural network - Keras

```
from keras.models import Sequential, Model

# Predict and compare to true labels
score = model.evaluate(Xtest,Ytest,batch_size=100)

# Predict only
predictions = model.predict(Xtest,batch_size=100)
```

See https://keras.io/models/model/#evaluate and https://keras.io/models/model/#predict

Neural networks - hyperparameters

- Optimization algorithm (and its parameters)
- Learning rate
- (Mini)batch size
- Number of hidden layers
- Number of nodes per hidden layer
- Parameters for regularization (λ_1, λ_2, dropout rate)

Split the data

- Split the data into training, validation and test
  (e.g. 70%, 15%, 15%)
- Training data: to learn parameters in network
- Validation data: estimate generalization error,
  used to tune hyper parameters
- Test data: estimate generalization error,
  not used for training or hyper parameter tuning
- For increased confusion,
  some call test data for validation data and vice versa...

Tensorflow playground

- Play around with a neural network in your browser, playground.tensorflow.org

Outline - part 2 of 6

- Deep learning, why?
- Types of networks and learning
- Deep learning basics
  - Activation functions
  - Initialization of weights
  - Optimization methods
  - Batch normalization

Why deep learning?

- To obtain better results compared to other techniques
- Deep networks can model more complex functions, compared to shallow networks (due to more parameters / non-linearities)
- Drawback: harder to understand what the network is doing
- No clear definition of what a "deep" network is
- Why now?
  - Bigger datasets
  - Faster computers (graphics cards)
  - Better optimization methods

Deep learning network types

- Your data determines what type of network to use
- General data without temporal / spatial correlation
  - (Deep) Neural Networks
- Sound (timeseries) / images / volumes
  - Convolutional Neural Networks (1D/2D/3D)
- (Temporal) sequences, text, sound, genetics
  - Recurrent Neural Networks
- Data / image synthesis
  - Generative Adversarial Networks

Types of deep learning

- Supervised learning, requires large amounts of annotated training data
  - Classification (predict class), regression (predict numerical value)
- Unsupervised learning, no labels / annotations required
  - Clustering, generative adversarial networks
- Reinforcement learning, no fixed training dataset, goal(s) and actions
- Semi-supervised learning, some annotated data, mostly un-annotated data

Activation functions

- Before deep learning, most neural networks used sigmoid or hyperbolic tangent as activation functions
- In deep learning, ReLU (rectified linear unit) and Leaky ReLU are most common
- ReLU: $y = max(0, x)$
- For ReLU, the gradient is always large for positive activations (i.e. no vanishing gradients), but 0 for negative activations
- Leaky ReLU: αx for $x < 0$, x for $x >= 0$, hyperparameter α
- For Leaky Relu, the gradient is only 0 if $x = 0$

ReLU vs leaky (parametric) ReLU

Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. IEEE international conference on computer vision (pp. 1026-1034).

Activation functions - gradients

For ReLU, the gradient is never zero for activations > 0

Initialization of weights

- Never initialize all weights to 0! Gives a linear model
- Before deep learning, the weights in a neural network were often initialized using random numbers from standard Gaussian (mean 0, variance 1)
- In deep learning, the initialization is more important due to many layers in a network (numerical stability) (and also depends on the activation function)
- Too large weights: exploding gradients
- Too small weights: vanishing gradients

Initialization of weights

- Initalization adapted to layer with m inputs, n outputs

  $W_{i,j} = U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$
- Xavier (Glorot) initialization (for tanh),

  $W_{i,j} = U(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{1}{m+n}})$
- He initialization (for ReLU),

  $W_{i,j} = \sqrt{\frac{2}{m}} \cdot N(0,1)$

- See https://keras.io/initializers/ for details

Working with (mini)batches

- Can normally not fit all training data in GPU memory
- Also need to store weights of network,
  and outputs / filter responses for each layer
- Need to work on batches of a smaller number of training examples,
  in some cases batch size is 1 (e.g. single image / volume)
- Most extreme case, cannot fit a single training example on GPU,
  digital pathology; image can be 50 000 x 50 000 pixels,
  need to work on patches / subvolumes
- Problem with small batches: gradient is more noisy

Optimization methods

- Since deep networks have a large number of parameters, the optimization methods become more important
- SGD, stochastic gradient descent, gradient descent on (mini)batch
- Momentum based approach, moving average of past gradients
- AdaGrad
- RMSprop (root mean square)
- Adam (adaptive momentum)

Optimization methods - momentum

- Smooths the gradient by updating weights using a combination of current and past gradients
- Moving average, $v = \gamma v + \nabla L(W)$
- Update parameters, $W = W - \alpha v$
- Hyperparameter γ, e.g. 0.9

Optimization methods - adagrad

- No momentum term, keep sum of squared gradients
- Updates in any direction are made with the same proportions (down-scale steep gradients, upscale small gradients)
- Accumulate squared gradient, $r = r + \nabla L(W)^2$
- Update parameters, $W = W - \alpha \frac{\nabla L(W)}{\sqrt{r} + \epsilon}$

Optimization methods - RMSprop

- Similar to AdaGrad, but uses a weighted moving average instead of summing all gradients
  (learning rate will become too low for AdaGrad)
- Accumulate squared gradient, $r = \rho r + (1 - \rho)\nabla L(W)^2$
- Update parameters, $W = W - \alpha \frac{\nabla L(W)}{\sqrt{r} + \epsilon}$
- Hyperparameter ρ, e.g. 0.9

Optimization methods - adam

- Combines the ideas of momentum and scaling the gradient
- Momentum, $m = \beta_1 m + (1 - \beta_1)\nabla L(W))$
- Accumulate squared gradient, $r = r + \beta_2 r + (1 - \beta_2)\nabla L(W)^2$
- Update weights, $W = W - \alpha \frac{m}{\sqrt{r} + \epsilon}$
- Default hyper parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ works in most cases
- Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Optimization methods - comparison

- See visual comparison at https://ruder.io/optimizing-gradient-descent/

Batch normalization

- A technique for improving speed, performance and stability when training deep neural networks, not yet fully understood
- Batch B with m training examples,
  calculate mean and variance (per dimension)
- $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$, $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$
- Normalize each dimension k separately
- $\hat{x}_i^k = \frac{x_i^k - \mu_B^k}{\sqrt{\sigma_B^{k2} + \epsilon}}$
- $\hat{x}^k$ now has zero mean and unit variance
- See book for details

Batch normalization

Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Ioffe, Sergey; Szegedy, Christian (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". arXiv:1502.03167

Batch normalization - Keras

```
from keras.models import Sequential, Model
from keras.layers.normalization import BatchNormalizat

model = Sequential()
model.add(Dense(10))
model.add(BatchNormalization())
```

See https://keras.io/layers/normalization/

keras-contrib provides instance normalization

Outline - part 3 of 6

- Convolutional neural networks (CNNs)
  - Convolutional layers, pooling layers, ...
- Image classification
  - AlexNet, GoogLeNet, ResNet, DenseNet, ...
- Image segmentation
  - U-Net, U-Net + VAE
- Multi-task learning (classification + segmentation)

Image classification

- In computer vision
  - What kind of object is present in an image? Many classes
- In medical imaging
  - Is the person sick or healthy? Binary
  - What is the disease? Many classes
  - What type of cancer is present?

Image classification - 2D example, CV

- ImageNet dataset / challenge, 1.2 million training images, 1000 classes
- http://www.image-net.org
-

- https://devopedia.org/imagenet

Image classification - 2D example, MI

Our model, CheXNet, is a 121-layer convolutional neural network that inputs a chest X-ray image and outputs the probability of pneumonia along with a heatmap localizing the areas of the image most indicative of pneumonia.

We train CheXNet on the recently released ChestX-ray14 dataset, which contains 112,120 frontal-view chest X-ray images individually labeled with up to 14 different thoracic diseases, including pneumonia. We use dense connections and batch normalization to make the optimization of such a deep network tractable.

Rajpurkar et al. (2017). Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. arXiv preprint arXiv:1711.05225.

Image classification - 3D example, MI

ELSEVIER

NeuroImage

Volume 145, Part B, 15 January 2017, Pages 137-165

Single subject prediction of brain disorders in neuroimaging: Promises and pitfalls

Mohammad R. Arbabshirani [a, b], Sergey Plis [a], Jing Sui [a, c], Vince D. Calhoun [a, d]

Show more

https://doi.org/10.1016/j.neuroimage.2016.02.079

Get rights and content

Highlights

- Past efforts on classification of brain disorders are comprehensively reviewed.
- The common pitfalls from machine learning point of view are discussed.
- Emerging trends related to single-subject prediction are reviewed and discussed.

Image regression - 3D example, MI

ELSEVIER

NeuroImage
Volume 163, December 2017, Pages 115-124

Predicting brain age with deep learning from raw imaging data results in a reliable and heritable biomarker

James H. Cole [a], Rudra P.K. Poudel [b], Dimosthenis Tsagkrasoulis [c], Matthan W.A. Caan [d], Claire Steves [e], Tim D. Spector [e], Giovanni Montana [b, c]

Show more

https://doi.org/10.1016/j.neuroimage.2017.07.059

Get rights and content

Highlights

- Chronological age can be accurately predicted using convolutional neural networks.
- Age predicted is accurate even using raw structural neuroimaging data.
- Brain-predicted age can be generated in a clinically applicable timeframe.
- Brain-predicted age is significantly heritable.
- Brain-predicted age is highly reliable, both within and between scanners.

Why deep learning?

- Before deep learning,
  most researchers worked on hand made features
- Instead of feeding a raw image to the network,
  extract features such as edges, lines, corners
- CNNs became very popular in 2012,
  when a 5-layer CNN outperformed other methods on ImageNet
- Krizhevsky, A., Sutskever, I., Hinton, G., ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012
- 47 900 citations!

Why convolution?

- Why use convolution instead of a standard deep neural network?
  - To lower the number of parameters to train
  - To use the local spatial correlation in the data
- A one megapixel image (1000 x 1000) has 1 000 000 "dimensions"
- For a single neural network layer with 100 nodes,
  need to train 100 000 000 parameters / weights for standard network
- 100 filters of size 3 x 3,
  1000 parameters / weights per layer for CNN (900 + 100 bias)

Standard CNN architecture

Figure 2.4: *Illustration of the data flow through the network VGG16. Data size, of the format* $[c, h, w]$*, is shown for the input image, output of each max-pooling layer, output after the first two fully connected layers, and the final network output.*

Jesper Westell, Multi-Task Learning using Road Surface Condition Classification and Road Scene Semantic Segmentation, LIU-IMT-TFK-A–19/570–SE

Real even (top) and imaginary odd (bottom) optimized quadrature filters in spatial domain - different directions

Quadrature filter responses, magnitude - different directions

Filter 1

Filter 2

Filter 3

Filter 4

Convolutional layer

- Every 2D convolutional layer takes a 4D tensor
  and returns another 4D tensor
- The 4 dimensions are batch_size, height, width, channels
  (or batch_size, channels, height, width)
- You set the number of filters to be used in every Conv2D layer
  (gives the number of channels in output tensor), filter size, padding etc
- A 1D convolutional layer works in the same way, but with 3D tensors
- A 3D convolutional layer works in the same way, but with 5D tensors

Conv2D is 3D, sort of

- Conv2D performs 2D convolution over the image, but each filter also learns how to linearly combine all channels in the input tensor
- This is not a true 3D convolution, but sort of
- Filter size 5 x 5 and 3 input channels, the learned filter is 5 x 5 x 3
- Filter 1: 0.25 * channel 1 + 0.8 * channel 2 ...
- Filter 2: -0.4 * channel 1 + 0.2 * channel 2 ...

Convolutional layer - Keras

```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D

model = Sequential()

# Layer 1
model.add(Conv2D(filters=32,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu',
                 input_shape=input_shape))
# Layer 2
model.add(Conv2D(filters=64,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu')
```

See https://keras.io/layers/convolutional/ for details

Finding the best filters

- In classic image processing,
  the filters are designed manually, filter optimization
  (e.g. to detect lines and edges of a certain size/frequency)

- In deep learning,
  the filters are learned through back propagation

- Benefit of deep learning,
  can learn very advanced filters, specific for your data

- Drawback of deep learning,
  need to perform convolution with many filters during many iterations
  (time consuming, memory demanding)

Convolution as matrix vector operation

- Convolution is a linear operation,
  all linear operations can be written as (sparse) matrix multiplications
  (see Nx x Ny image / matrix as (Nx · Ny) x 1 vector)

Figure 2.1: Left: $\frac{1}{4}[1, 2, 1]$ filter matrix. Right: $[1, 0, -1]$ filter matrix.

- G. Johansson, A Global Linear Optimization Framework
  for Adaptive Filtering and Image Registration, 2015

- Sparse matrices and parameter sharing (same filter everywhere)

Convolution as matrix vector operation

Convolutional network

Unshared convolution / Locally connected layers

Fully connected network

Convolution as matrix vector operation

- Every image in batch is represented as a long vector
- Put all images in batch into a matrix X
- Apply filter W in a layer to all images using $WX + B$
- Operations performed by CUDNN, details are secret
- We can re-use all methods / algorithms for neural networks!

Handling the border of the image

- Values outside each image are undefined
- Solution 1, only calculate valid filter response,
  filter response will be (Nf - 1) pixels smaller (Nf size of filter)
- Solution 2, assume values are 0 (zero padding),
  filter response will be same size as image
- Why is this important?
- For a filter of size 7 x 7, the image will be 6 x 6 smaller
  in every layer of the network for solution 1
  (and eventually we run out of pixels)

Convolutional layer - padding

- Input 4D tensor of size (batch_size, height, width, channels)
- 64 filters of size 5 x 5
- 'same' padding,
  - output 4D tensor is of size (batch_size, height, width, 64)
- 'valid' padding,
  - output 4D tensor is of size (batch_size, height - 4, width - 4, 64)

Pooling layers

- Pooling is normally done in every (second) layer of a CNN
- Most common, max or average operation over a small neighbourhood (pixels/voxels)
- Without pooling, the data would have the same size in every layer (and the final fully connected layer would be very large)
- Can do pooling over filter responses in each pixel/voxel
- Pooling can be used to become invariant to
  - Translation / shift
  - Scale ?
  - Rotation (Figure 9.9)

Pooling - rotation invariance

Figure 9.9: Example of learned invariances. A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input. Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand written 5.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

Pooling layer - Keras

```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, MaxPooling2D

model = Sequential()

model.add(Conv2D(filters=32,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu',
                 input_shape=input_shape))

model.add(MaxPooling2D(pool_size=(2, 2)))
```

See https://keras.io/layers/pooling/ for details

Pooling - reduce number of parameters

	Activation shape	Activation size	# parameters
Input	(32,32,3)	3072	0
Conv2D (8 filters, size 5)	(28,28,8)	6272	608
MaxPooling2D (2,2)	(14,14,8)	1568	0
Conv2D (16 filters, size 5)	(10,10,16)	1600	3216
MaxPooling2D (2,2)	(5,5,16)	400	0
Fully Connected (120 nodes)	(120,1)	120	48001
Fully Connected (84 nodes)	(84,1)	84	10081
Softmax	(10,1)	10	841

First layer, 8 filters of size 5 x 5 x 3, + 8 biases = 608 parameters

Second layer, 16 filters of size 5 x 5 x 8, + 16 biases = 3216 parameters

Fully connected layer 1, 400 * 120 + 1 = 48 001 parameters

Fully connected layer 2, 120 * 84 + 1 = 10 081 parameters

Fully connected layers

- After a number of layers with convolution and max-pooling, flatten tensor to a long vector, connect to fully connected layer(s) (standard neural network)
- Note: This will put a restriction on the image input size
- Fully convolutional networks can be trained on images of size A, and then be applied on images of size B
- No dense layers, all layers are convolutional
- 1 x 1 convolution, just combine all channels (features) into one or more scalars (same combination for all pixels)

Fully connected layer - Keras

```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, MaxPooling2D
from keras.layers import Flatten, Dense

model = Sequential()
model.add(Conv2D(filters=32,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten)
model.add(Dense(numClasses, activation='softmax'))
```

See https://keras.io/layers/core/ for details about Flatten and Dense

Adding regularization - Keras

```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D
from keras import regularizers as reg

model = Sequential()

model.add(Conv2D(filters=32,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu',
                 kernel_regularizer=reg.l1(0.001))

model.add(Conv2D(filters=32,kernel_size=(5,5),
                 padding = 'same',
                 activation='relu',
                 kernel_regularizer=reg.l2(0.001))
```

See https://keras.io/regularizers/ for details

Adding dropout - Keras

```
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Dropout

model = Sequential()

model.add(Dense(32, activation='sigmoid',
                input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Types of regularization & constraints

- kernel_regularizer, kernel weight matrix (filter)
- bias_regularizer, bias vector
- activity_regularizer, output of the layer
- kernel_constraint, for filter
- bias_constraint, for bias
- See https://keras.io/constraints/

Derivatives

- What about the derivatives needed for back propagation?
- Difficult to keep track of derivatives for 100 layers...
- Tensorflow / Keras will do all the work for you!
  (even if you add regularization and / or dropout)

Popular 2D CNN classification architectures

- LeNet-5 (1998), 7 layer CNN
- AlexNet (2012), 5 convolutional layers, 3 fully connected layers
- GoogLeNet (Inception V1) (2014), 22 layer CNN
- VGGNet (2014), 16 convolutional layers
- ResNet (2015), 152 convolutional layers, skip connections
- DenseNet (2017), dense connections

GoogLeNet (Inception V1)

- Promoted the idea of stacking the layers in CNNs more creatively, as networks in networks. Inside a relatively standard architecture (called the stem), GoogLeNet contains multiple inception modules, in which multiple different filter sizes are applied to the input and their results concatenated. This multi-scale processing allows the module to extract features at different levels of detail simultaneously.

- GoogLeNet also popularized the idea of not using fully-connected layers at the end, but rather global average pooling, significantly reducing the number of model parameters.

- Szegedy et al., Going deeper with convolutions, IEEE conference on computer vision and pattern recognition, 2015

GoogLeNet (Inception V1)

(a) Inception module, naïve version

(b) Inception module with dimensionality reduction

Figure 2: Inception module

VGGNet

- Popularized the idea of using smaller filter kernels and therefore deeper networks (up to 19 layers for VGG19, compared to 7 for AlexNet)
- Trained the deeper networks using pre-training on shallower versions
- Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556

ResNet

- Noticed that performance on training data dropped when having too many layers
- Introduced skip connections, which makes it possible to train much deeper networks. Some features are best constructed in shallow networks, while others require more depth. The skip connections facilitate both at the same time, increasing the network's flexibility when fed input data.
- He et al., Deep residual learning for image recognition, IEEE conference on computer vision and pattern recognition, 2016

ResNet - skip connections

DenseNet

- Builds on the ideas of ResNet, but instead of adding the activations produced by one layer to later layers, they are simply concatenated together. The original inputs in addition to the activations from previous layers are therefore kept at each layer, preserving some kind of global state. This encourages feature reuse and lowers the number of parameters for a given depth.
- DenseNets are therefore particularly well-suited for smaller data sets.
- Huang et al., Densely connected convolutional networks, IEEE conference on computer vision and pattern recognition, 2017

DENSENET

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

Pretrained 2D architectures

- Several popular 2D architectures are available as pretrained networks (for example trained on ImageNet)
- Instead of starting training from scratch, start from pretrained network
- Most of these pretrained networks are trained for color images, not for grayscale images (most common in medical imaging)
- Not so many pretrained networks available for 3D (Nvidia Clara ?)

Pretrained 2D architectures - Keras

Applications

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

Weights are downloaded automatically when instantiating a model. They are stored at `~/.keras/models/`.

Available models

Models for image classification with weights trained on ImageNet:

- Xception
- VGG16
- VGG19
- ResNet, ResNetV2
- InceptionV3
- InceptionResNetV2
- MobileNet
- MobileNetV2
- DenseNet
- NASNet

See https://keras.io/applications/

Pretrained 2D architectures - Keras

```
from keras.models import Sequential, Model
from keras.applications.vgg16 import VGG16

model = Sequential()
model.add(VGG16(weights='imagenet'))
model.summary()
```

Changing default classifier - Keras

```
from keras.models import Sequential, Model
from keras.applications.vgg16 import VGG16

model = Sequential()
model.add(VGG16(weights='imagenet',
                include_top=False,
                input_shape=(224,224,3)))
model.add(Flatten())
model.add(Dense10)
```

Freezing layers in pretrained models

- Pretrained models are good for initialization
- Some parts of the network still need to be trained,
  to fit your specific data, to improve performance
- Early convolutional layers learn basic features (edges, lines, ...)
- Freeze early (top) layers, train deep (bottom) layers

Freezing layers - Keras

```
from keras.models import Sequential, Model
from keras.applications.vgg16 import VGG16

model = Sequential()
model.add(VGG16(weights='imagenet',
                include_top=False,
                input_shape=(224,224,3)))

# Freeze the first 10 layers
for layer in VGG16.layers[:10]:
    layer.trainable = False
```

What about 3D CNNs?

- Easy to change Tensorflow code from Conv2D to Conv3D
- MaxPooling2D -> MaxPooling3D
- Will increase graphics memory usage,
  cannot use as many filters per layer
- Will increase processing time
- Keras has built-in 2D data augmentation, but not for 3D

Image segmentation

- Compared to image classification,
  image segmentation can be seen as classification of every pixel / voxel
- The output is not a scalar per image, but a new image
- Each pixel / voxel is one training example,
  less training data required compared to image classification

Applications - scene segmentation

-
- https://www.youtube.com/watch?v=PNzQ4PNZSzc

Applications - brain tumour segmentation

Fig. 6. Progression of learning in InputCascadeCNN*. The stream of figures on the first row from left to right show the learning process during the first phase. As the model learns better features, it can better distinguish boundaries between tumor sub-classes. This is made possible due to uniform label distribution of patches during the first phase training which makes the model believe all classes are equiprobable and causes some false positives. This drawback is alleviated by training a second phase (shown in second row from left to right) on a distribution closer to the true distribution of labels. The color codes are as follows: edema, enhanced tumor, necrosis, non-enhanced tumor.

Training data for image segmentation - binary

Manual segmentation (time consuming)
Use some other segmentation algorithm, and edit segmentations

Loss functions for image segmentation

- Binary segmentation, binary cross entropy
- Weighted cross entropy, to handle class imbalance,

  $L = -(\beta\, t\, logy + (1-t) log(1-y))$
- To decrease false negatives, use $\beta > 1$,
  to decrease false positives, use $\beta < 1$
  (Tensorflow, weighted_cross_entropy_with_logits)
- Multi-class segmentation, categorical cross entropy

Loss functions for image segmentation

- Segmentation results are often evaluated using Dice score, but not so meaningful to calculate Dice for single pixel

  $$Dice = 1 - \frac{2\,t\,y+1}{y+t+1}$$

- Global Dice loss,

  $$Dice = 1 - \frac{2\sum_{pixels} t\,y}{\sum_{pixels} y + \sum_{pixels} t}$$

- Combinations such as cross-entropy (local) + Dice (global)

Segmentation architectures - U-Net

Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Consists of encoder (standard CNN), bridge, and decoder (upsampling)
Encoder uses Conv2D, decoder uses Conv2DTranspose
Uses skip connections, inspired by ResNet

Ronneberger et al., U-Net: Convolutional Networks for Biomedical Image Segmentation, arXiv:1505.04597

Conv2DTranspose layer

- Transposed convolution layer (sometimes called Deconvolution).
- The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

Segmentation architectures - U-Net

- Original U-Net paper has 9600 citations
- Most new segmentation architectures are variants of U-Net
- Cascaded U-Net, several U-Nets after each other
- Adding a variational autoencoder for regularization

U-NET + VAE

The encoder-decoder architecture employed in Myronenko (2018). The top decoder branch produces the tumor segmentation maps, while the bottom one reconstructs the input volume.

Myronenko, Andriy. "3D MRI brain tumor segmentation using autoencoder regularization." International MICCAI Brainlesion Workshop. Springer, Cham, 2018

Multi-task learning (MTL)

- Train a network for several tasks; classification and segmentation
- Shared feature extractor (e.g. VGG16, ResNet-101, ...)
- Forcing a feature extractor to be good at several tasks
  can lead to better regularization => better performance

Multi-task learning (MTL)

Figure 3.5: *MTL network architecture.*

Jesper Westell, Multi-Task Learning using Road Surface Condition Classification and Road Scene Semantic Segmentation, LIU-IMT-TFK-A–19/570–SE