2017: Overview of state-of-art of computer vision

A Survey on Deep Learning in Medical Image Analysis

(great overview of the state of the art)

Introduction

Initially, from the 1970s to the 1990s, medical image analysis was done with sequential application of low-level pixel processing (edge and line detector filters, region growing) and mathematical modeling (fitting lines, circles and ellipses) to construct compound rule-based systems that solved particular tasks. Heavy reliance on ***if-then-else statements*** and known as GOFAI (good old-fashioned artificial intelligence).

At the end of the 1990s, supervised techniques, where training data is used to develop a system, were becoming increasingly popular in medical image analysis. Examples include active shape models (for segmentation) and the concept of feature extraction and use of statistical classifiers (for computer-aided detection and diagnosis). This pattern recognition or machine learning approach is still very popular today and forms the basis of many successful commercially available image analysis systems. Thus we have seen a shift from systems that are completely designed by humans to systems that are trained by computers using example data from which feature vectors are extracted. Computer algorithms determine the optimal decision boundary in the high-dimensional feature space. A crucial step in the design of such systems is the extraction of discriminant features from the images. This process is still done by human researchers and, as such, one speaks of systems with ***handcrafted*** features.

A logical next step is to ***let computers learn the features*** that optimally represent the data for the problem at hand. This concept lies at the basis of many ***deep learning algorithms***: models (networks) composed of many ***layers*** that transform input data, e.g. images, to outputs, e.g. disease present/absent, while learning increasingly higher level features. The most successful type of models for image analysis to date are ***convolutional neural networks (CNN)***. CNNs contain many layers that transform their input with convolutional filters to a small extent.

Work on CNNs has been done since the late seventies (Fukushima, 1980) and they saw their first successful real-world application in ***LeNet*** (leCun et al., 1998) for hand-written digit recognition but the watershed came with Krizhevsky et al's ***AlexNet winning the 2012 ImageNet challenge*** by a large margin. ***In computer vision, deep convolutional networks have now become the technique of choice***.

Overview of deep learning methods

Learning Algorithms

Machine Learning methods are generally divided into ***supervised and unsupervised*** learning algorithms, although there are many nuances. In ***supervised learning***, a model is represented by a dataset of input features, x, and label, y, pairs, where y typically represents an instance of a fixed set

of classes. Supervised training typically amounts to ***finding model parameters, Θ***, that best ***predict*** the data based on a ***loss function***.

Unsupervised learning algorithms process data ***without labels*** and are trained to find patterns, such as latent subspaces. Examples of traditional unsupervised learning algorithms are ***Principle Componant Analysis*** and ***Clustering*** methods.

Neural Networks

Neural networks are a type of learning algorithm which forms the basis of most deep learning methods. A neural network comprises of neurons or units with:

- some ***activation, a,*** and
- ***parameters theta = {W,B}*** where W is a set of weights and B is a set of biases

The activation represents a ***linear combination*** of the input x to the neuron and the parameters, followed by an element-wise non-linearity σ(.), referred as a transfer function.

$$A = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

Typical transfer functions for traditional neural networks are the sigmoid and hyperbolic tangent function. The **multi-layered perceptrons (MLP)**, the most well-known of the traditional neural networks, have several layers of these transformations:

$$f(\mathbf{x}; \mathbf{\Theta}) = \sigma(\mathbf{W}^T \sigma(\mathbf{W}^T ...\sigma(\mathbf{W}^T\mathbf{x} + b)) + b)$$

Here, **W** is a matrix comprising of columns $\mathbf{w}_k$, associated with activation k in the output. Layers in between the input and output are often referred to as '***hidden***' layers. When a neural network contains ***multiple hidden layers*** it is typically considered a '***deep' neural network***, hence the term ***'deep learning'***.

At the final layer of the network the activations are mapped to a distribution over classes $P(y|\mathbf{x}; \Theta)$ through a softmax function.

$$P(y|\mathbf{x}; \mathbf{\Theta}) = \text{softmax}(\mathbf{x}; \mathbf{\Theta})$$

Maximum Likelihood with stochastic gradient descent is currently the most popular method to fit parameters Θ to a dataset D. In stochastic gradient descent a small subset of the data, a mini-batch, is used for each gradient update instead of the full data set. Optimising maximum likelihood in practice amounts to minimising the negative log-likelihood → ***descibed in detail in my Columbia Machine Learning notes***.

Currently, the most popular models are trained end-to-end in a supervised fashion, greatly simplifying the training process. The most popular architectures are convolutional neural networks (CNN), widely used in image analysis, and recurrent neural networks (RNN).

Convolutional Neural Networks

There are two key differences between MLPs and CNNs. First, in CNNs weights in the network are shared, i.e. parameter sharing, in such a way that it is the network that performs convolution operations on the images.

The second key difference is the typical incorporation of pooling layers in CNNs, where pixel values of neighbourhoods are aggregated using a permutation invariant function, typically the max or mean operation.

At the end of the convolutional stream of the network, fully-connected layers, i.e regular neural network layers, are usually added, where weights are no longer shared. Similar to MLPs, a distribution over classes is generated by feeding the activations in the final layer through a softmax function and the network is trained using maximum likelihood.

Deep CNN Architectures

(Taken from Convolutional Neural Networks)

- ***LeNet***
  First successful application of Convolutional Networks
  Developed by Yann LeCun in 1990s
  Used to read zip codes, digits, etc.

- ***AlexNet***
  First work to popularise Convolutional Networks on Computer Vision
  Developed by Alex Krizhevsky, Ilya Sutskever & Geoff Hinton
  Won ImageNet ILSVRC challenge in 2012
  Similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other, rather than a single CONV layer immediately followed by a POOLING layer.

- ***ZF Net***
  Matthew Zeiler & Rob Fergus won 2013 ImageNet ILSVRC challenge with this architecture
  Improved upon AlexNet by tweaking architecture hyper-parameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.

- ***GoogLeNet***
  Szegedy et al from Google won 2014 ImageNet ILSVRC challenge with this architecture
  Main contribution was the development of an Inception Module, most recently Inception v4, which dramatically reduced the number of parameters in the network (4M, compared to AlexNet's 64M).
  Uses Average Pooling instead of fully connected layers at the top of the ConvNet, eliminating lots of parameters that don't seem to matter

- ***VGGNet***
  Karen Simonyan and Andrew Zisserman came 2nd in 2014.
  Main contribution was in showing that the depth of the network is a critical component for good performance.

- ***ResNet***
  Kaiming et al won 2015 challenge
  Features special skipped connections, and a heavy use of batch normalisation.

Also missing fully connected layers at the end of the network
State of the art in Convolutional Neural Network models and are the default choice for using ConvNets in practice.
Recent developments that tweak the original architecture can be found in Identity mappings in deep residual networks by Kaming et al.

Recurrent Neural Networks (RNN)

Traditionally, RNNs were developed for discrete sequence analysis. They can be seen as a generalisation of MLPs because both the input and output can be of varying length, making them suitable for tasks such as machine translation where a sentence of the source and target language are the input and output. In a classification setting, the model learns a distribution over classes

$$P(y|\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T; \Theta)$$

given a sequence **x**1, **x**2, …,**x**T, rather than a single input vector **x**.

The most popular RNN is the Long Short Term Memory (LSTM).

Unsupervised Models

Includes:

- Auto-encoders (AEs) and Stacked Auto-encoders
- Restricted Boltzmann Machines (RBMs) and Deep Belief Networks (DBNs)
- Variational Auto-Encoders and Generative Adverserial Networks

Hardware and Software

One of the main contributions to the steep rise of deep learning has been the widespread availability of GPU and GPU-computing liraries (CUDA, OpenCL). GPUs are highly parallel computing engines, which have an order of magnitude more execution threads than central processing units (CPUs). With current hardware, deep learning on GPUs is typically 10 to 30 times faster than CPUs.

Next to hardware, the other driving force behind the popularity of deep learning methods is the wide availability of open source software packages. These libraries provide efficient GPU implementations of important operations in neural networks, such as convolutions; allowing the user to implement ideas at a high level rather than worrying about low-level efficient implementations. The most popular packages are:

- **Caffe**
  Provides C++ and Python interfaces,
  Developed at UC Berkeley
- **Tensorflow**
  Provides C++ and Python interfaces,
  Developed by Google

- **Theano**
  Provides a Python interface,
  Developed by MILA lab in Mntreal
- **Torch**
  Provides a Lua interface
  Used by Facebook research

Other libraries of interest are Lasange and Keras.

2016: Interface that abstracts away computational & mathematical complexities

TensorFlow: A system for large-scale machine learning

(Together with tensorflow web site, the main training source for using TensorFlow)

Abstract

TensorFlow is a a machine learning system that operates at large scale and in heterogeneous environments. TensorFlow uses ***dataflow graphs*** to represent computation, shared state, and the operations that mutate the state. It maps the nodes of a dataflow graph across many machines in a cluster, and within a machine across multiple computational devices, including multicore GPUs, and custom designed ASICs known as tensor Processing Units (TPUs). This architecture gives flexibility to the application developer: whereas in previous "parameter server" designs the management of shared state is built into the system, TensorFlow enables developers to experiment with novel optimisations and training algorithms. TensorFlow supports a variety of applications, with a focus on training and inference on deep neural networks. Several Google services use TensorFlow in production, we have released it as an open-source project, and it has become widely used for machine learning research. In this paper, we descibe the TensorFlow dataflow model and demonstrate the compelling performance that TensorFlow achieves for several real-world applications.

Introduction

Background & Motivation

Previous system: DistBelief

Design Principles

- Dataflow graphs of primitive operators
- Deferred execution
- Common abstraction for heterogeneous accelerators

Related Work

- Single-machine frameworks
- Batch dataflow systems
- Parameter servers

TensorFlow execution model

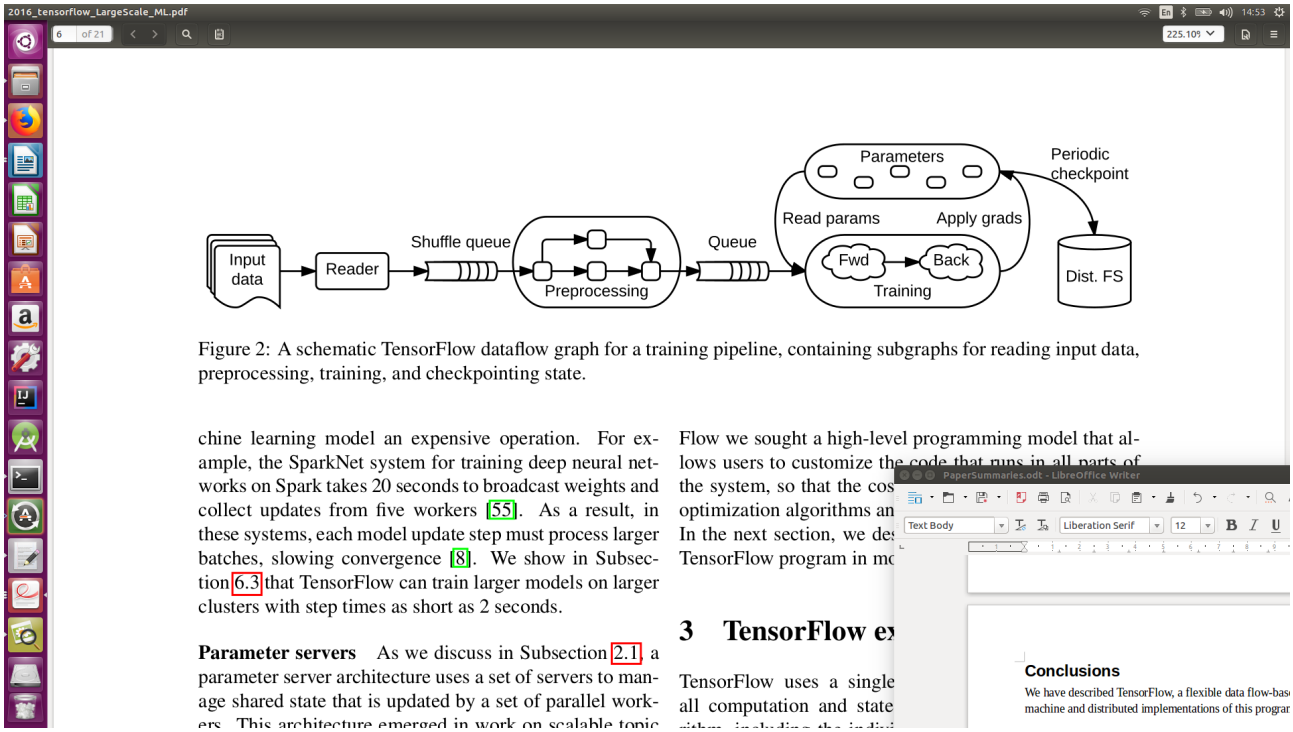


Figure 2: A schematic TensorFlow dataflow graph for a training pipeline, containing subgraphs for reading input data, preprocessing, training, and checkpointing state.

chine learning model an expensive operation. For example, the SparkNet system for training deep neural networks on Spark takes 20 seconds to broadcast weights and collect updates from five workers [55]. As a result, in these systems, each model update step must process larger batches, slowing convergence [8]. We show in Subsection 6.3 that TensorFlow can train larger models on larger clusters with step times as short as 2 seconds.

Parameter servers As we discuss in Subsection 2.1, a parameter server architecture uses a set of servers to manage shared state that is updated by a set of parallel workers. This architecture emerged in work on scalable topic

Flow we sought a high-level programming model that allows users to customize the code that runs in all parts of the system, so that the cos optimization algorithms an In the next section, we des TensorFlow program in mo

3 TensorFlow ex

TensorFlow uses a single all computation and state

Illustration 1: TensorFlow execution model

Dataflow graph elements

- Tensors
- Operations
- Stateful operations: variables
- Stateful operations: queues

Partial and concurrent execution

Distributed execution

Dynamic control flow

Extensibility case studies

Differentiation and optimisation

Training very large models

Fault tolerance

Synchronous replica coordination

Implementation

Evaluation

Single-machine benchmarks

Synchronous replica microbenchmark

Image classification

I want to replicate this on my machine but download pre-baked Imagnet models and run evaluation and predictions for all major models:

- AlexNet
- ZF Net
- GoogleNet
- VGGNet
- ResNet

Language modeling

Conclusions

We have described the TensorFlow system and its programming model. TensorFlow's dataflow representation subsumes existing work on parameter server systems, and offers a set of uniform abstractions that allows users to harness large-scale heterogeneous systems, both for production tasks and for experimenting with new approaches. We have shown several examples of how the TensorFlow programming model facilitates experimemnation and demonstrated that the resulting implementations are performant and scalable.

2016: Interface that abstracts away parallel & distributed tasks

TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems

(main engineering paper)

Abstract

TensorFlow is an ***interface for expressing machine learning algorithms***, and an ***implementation for executing such algorithms***. A computation expressed using TensorFlow can be executed with little or no change on a wide variety of heterogeneous systems, ranging from mobile devices such as phones and tablets up to large-scale distributed systems of hunderds of machines and thousands of computational devices such as GPU cards. The system is flexible and can be used to express a wide variety of algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learning systems into production across more than a dozen areas of computer science and other fields, including speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery. This paper describes the TensorFlow interface and an implementation of that interface that we have built at Google. The TensorFlow API and a reference implementation were released as an open-source package under the Apache 2.0 license in November, 2015 and are available at www.tensorflow.org

Introduction

Programming Model and Basic Concepts

Implementation

Single Device Execution

Multi Device Execution

- Node Placement
- Cross Device Communication

Distributed Execution

Extensions

Gradient Computation

Partial Execution

Device Constraints

Control Flow

Input Operations

Queues

Containers

Optimisations

Common Subexpression Elimination

Controlling Data Communication and Memory Usage

Asynchronous Kernels

Optimised Libraries for Kernel Implementations

Lossy Compression

Status and Experience

Common Programming Idioms

Performance

Tools

TensorBoard: Visualisation of graph structures and summar statistics

Performance Tracing

Future Work

Related Work

Conclusions

We have described TensorFlow, a flexible data flow-based programming model, as well as single machine and distributed implementations of this programming model.

Convolutional Neural Networks (CNNs / ConvNets)

Source: https://cs231n.github.io/convolutional-networks/

Code Example: cnn_mnist.py

Introduction

- CNNs are very similar to ordinary Neural Networks.
- ConvNet architectures make the explicit assumption that the inputs are images,
- Allowing the encoding of certain properties into the architecture,
- Making the forward function more efficient to implement,
- Vastly reducing the number of ***parameters*** in the network.

Architecture Overview

- CNNs take advantage of an architectural constraint due to the input consisting of images.
- The layers of a ConvNet have neurons arranged in 3 dimensions:
  - width
  - height
  - depth (3rd dimension of an activation volume, not depth of a full neural network)

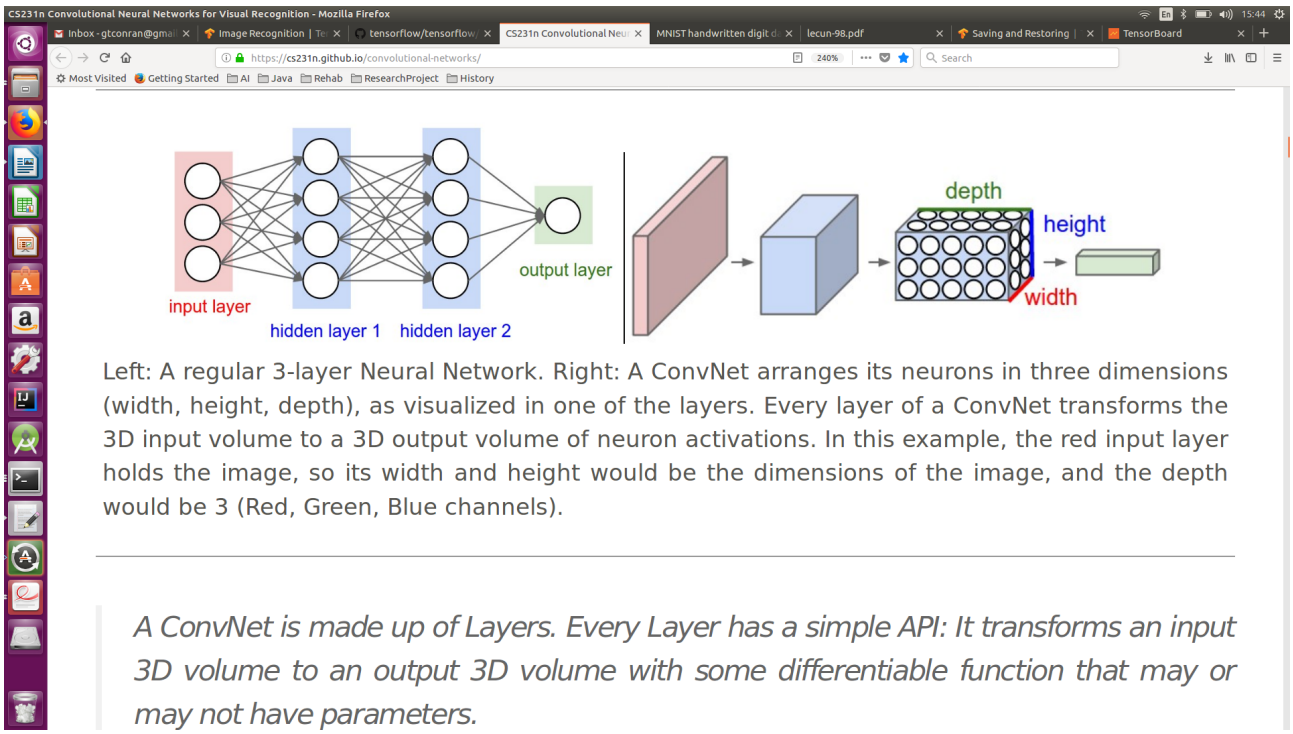


Illustration 2: ConvNet Architecture

Layers used to build ConvNets

As described earlier, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. Three main types of layers are used to build ConvNet architectures.

1. Convolutional layer
2. Pooling layer
3. Fully-connected layer

These layers are stacked to form a full Convnet architecture.

Example Architecture: overview

A simple ConvNet for CIFAR-10 classification could have the architecture:

INPUT → CONV → RELU → POOL → FC

In more detail (and with tensorflow code (High & Low level API examples):

- **INPUT layer**
  Holds the raw pixel values of the image, in this case an image of w:32, h:32, d:3
  High tf: input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
  Low tf: input_layer = tf.reshape(features[“x”], [-1, 28, 28, 1])
- **CONV layer**
  Computes the ouput of neurons that are ***connected to local regions*** in the input.
  Each computing a dot product between their weights and a small region they are connected to in the input volume
  This may result in volume such as [32x32x12] if we used 12 filters (or kernels)
  High tf: conv1 = tf.layers.conv2d(
  inputs=input_layer,
  filters=32,
  kernel_size=[5, 5],
  padding="same",
  activation=tf.nn.relu)
  Low tf: w1 = tf.Variable(
  tf.truncated_normal([patch_size,patch_size,image_depth,patch_depth], stddev=0.1))
  b1 = tf.Variable(tf.zeros([patch_depth1]))
  conv1= tf.nn.conv2d(input_layer, w1, [1,1,1,1], padding=’SAME’)
- **RELU layer**
  Applies an element wise activation function, such as max(x,0) thresholding at zero
  This leaves the size of the volume unchanged ([32x32x12])
  High tf: included in conv1 above
  Low tf: actv1 = tf.nn.relu(conv1 + w1 + b1)
- **POOL layer**
  Performs a ***downsampling operation*** along the spatial dimensions (width, height),
  Resulting in volume such as [16x16x12].
  High tf: pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
  Low t f: pool1 = tf.nn.max_pooling(actv1, [1,2,2,1], [1,2,2,1], padding=’SAME’)
- **FULLY CONNECTED (FC) layer**
  Computes the class scores, resulting in volume [1x1x10],
  Where each of the 10 numbers correspond to a class score, such as the 10 categories of MNIST.
  Each neuron in this layer will be connected to all the numbers in the previous volume.
  High tf: dense = tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
  logits = tf.layers.dense(inputs=dropout, units=10)
  Low tf: logits = tf.matmul(actv1, w1) + b1

In this way, ConvNets transform the original image layer by layer from the original pixels to the final class scores.

Note that some layers contain parameters and others don't:

- **CONV/FC layers**
  Perform transformations that are a function of the activations in the input volume AND parameters (the weights and biases of the neurons), parameters which are trained with gardient descent so that the class scores that the Convnet computes are consustent with the labels in the training set for each image.
- **RELU/POOL layers**
  Just implement a fixed function.

Convolutional Layer

Overview and intuition

- The core building block of a Convolutional Network --> computational heavy lifting.
- CONV parameters consist of a set of ***learnable filters***
- Filters are small spatially (along width & height) but extends the full length of input volume
- During ***forward pass***, we **convolve (slide)** each filter across the width & height of the input volume & compute dot products between entries of the filter and input at any position.
- As we slide the filter over the input volume we product a ***2-d activation map (feature map)*** that gives the r***esponses of that filter at every spatial position***.
- Intuitively, the ***network will learn filters that activate*** when they see some type of visual feature such as an edge or blotch of colour on the first layer, or eventually entire honeycomb or whell-like patterns on higher layers of the network.

Local Connectivity

- For high-dimensional inputs, e.g. images, it is impractical to connect neurons to all neurons in the previous volume.
- Instead, we connect each neuron to only a local region of the input volume.
- The spatial extent of this connectivity is a hyperparameter called a ***receptive field*** of the neuron (equivalently this is the filter size)
- The extent of the connectivity along the depth axis is always equal to the depth of the input volume, meaning the connections are local in space (along height & width), but always full along the entire depth of the input volume.

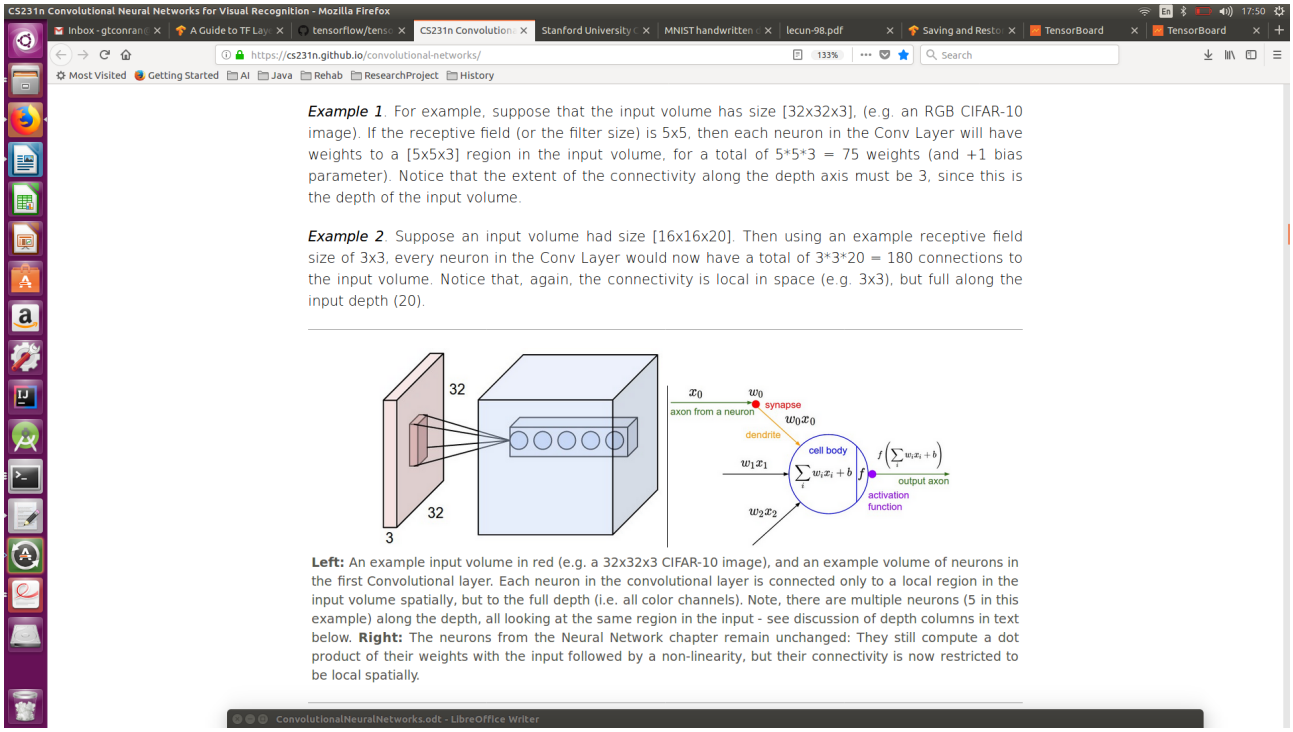


Illustration 3: Convolutional Layer Examples

Spatial Arrangement

A discussion on how many neurons there are in the ouput volume or how they are arranged. Three hyper-parameters control the size of the output volume:

- **Depth of the output volume**
  Corresponds to the number of filters we want to use, each learning to look for something different in the output.
  Depth column (fibre) is a set of neurons that are all looking at the same region of the input
- **Stride with which we slide the filter (kernel)**
  When 1 the filter is moved 1 pixel at a time. When 2, the filter jumps 2 pixels at a time, resulting in smaller output volumes spatially.
- **Zero-padding** → pad the input volume with zeros around the border
  Allows us to control the spatial size of the ouput volumes

The output volume can be computed as a function of:

- the input volume size, W =7x7
- the receptive field size of the Conv Layer neurons, F, = 3x3
- the stide with which they are applied, S = 1
- and the amount of zero padding used, P, = 0 on the border.

The formula being:

$$\mathbf{(W - F + 2P) / S + 1 = (7 - 3 + 2x0) / 1 + 1 = 5x5}$$

Parameter Sharing (key architectural feature of tensorflow)

Used in Convolutional Layers to control the number of parameters. The number of parameters can be dramatically reduced by making an assumption: if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute a different position (x2, y2).

If all neurons in a depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a ***convolution*** of the neuron's weights with the input volume, hence the name Covolutional Layer. This is why it is common to refer to the sets of weights as a ***filter*** (or a ***kernel***), that is convolved with the input.

Convolutional Layer: Summary

- Accepts a volume of size W1 x H1 x D1
- Requires four hyper-parameters
  - Number of filters, K
  - Their spatial extent, F
  - The Stride, S
  - The amount of zero-padding, P.
- Produces a volume of size W2 x H2 x D2 where:
  - W2 = (W1 – F +2P)/S +1
  - H2 = (H1 – F +2P)/2 +1
  - D2 = K
- With parameter sharing, it introduces F x F x D1 weights per filter, for a total of (FxFxD1)xK weights and K biases
- In the output volume, the d-th depth slice (of size W2 x H2) is the result of performing a volid convolution of the d-th filter over the input volume with a stride S, and then offset by d-th bias

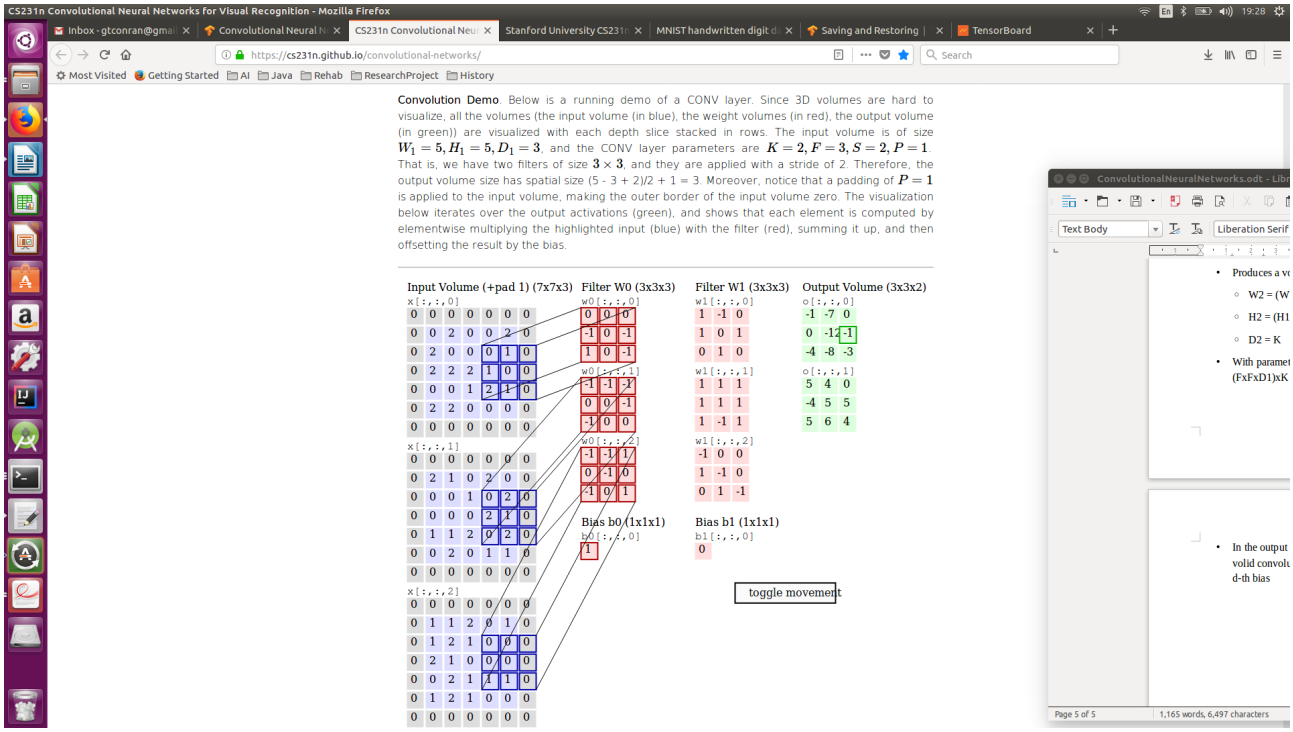


Illustration 4: Convolutional Demo

Pooling Layer

- It is common to periodically insert a Pooling layer in-between successive Conv layers
- Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to control over-fitting.
- Pooling layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.
- Terefore, the depth dimension remains the same

In general, the pooling layer:

- Accepts a volume of size W1 x H1 x D1
- Requires two hyper-parameters
  - their spatial extenf F,
  - the stide S,
- Produces a volume of size W2 x H2 x D2 where
  - W2 = (W1 – F)/S + 1
  - H2 = (H1 – F)/S +1
  - D2 = D1
- Introduces zero parameters since it computes a fixed function of the input

- Note that it is not common to use zero-padding for Pooling layers

Note, thetre are only 2 commonly seen variations of the max pooling layer found in practice:

- F=3, S=2 (called over-lapping pooling)
- F=2, S=2 (most common)

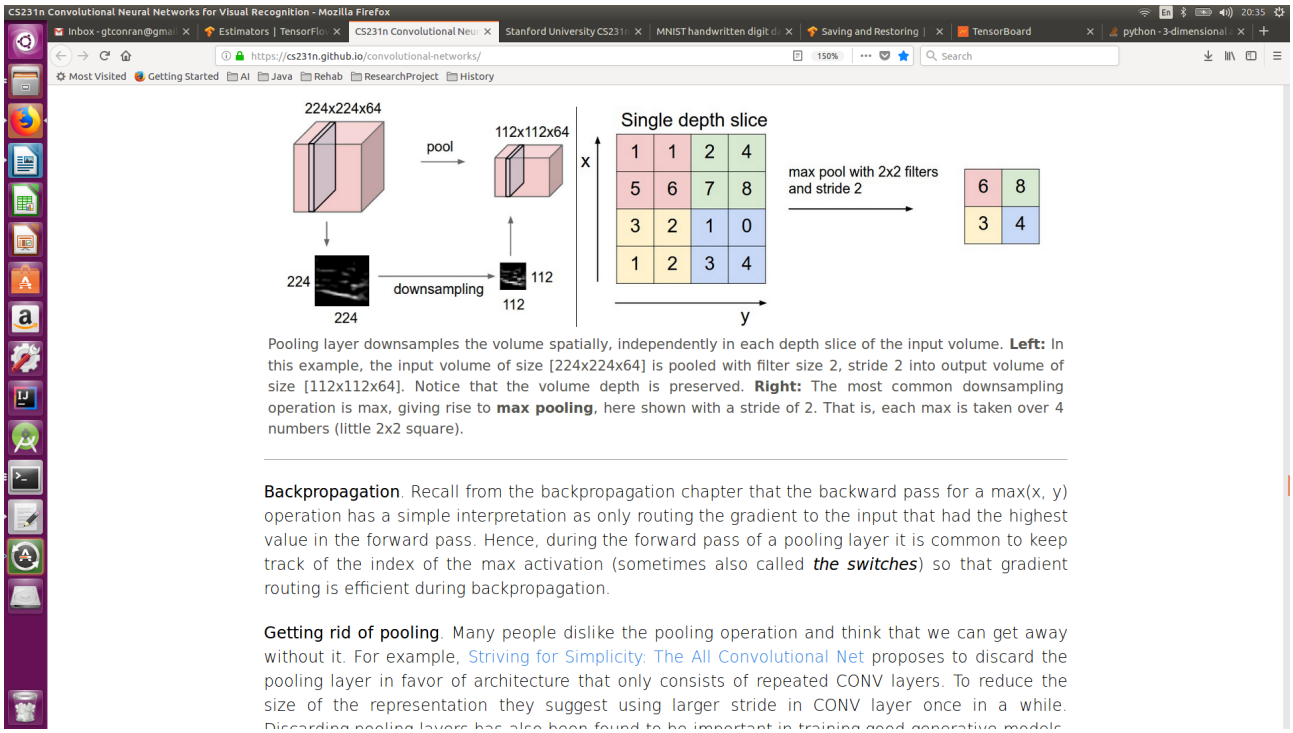


Illustration 5: Max Pooling

Fully Connected Layer

- Neurons in a fully connected layer have full connections to all activations in the previous layer
- Their activations ca be computed with a matrix multiplication followed by a bias offset
- Similatr to CONV layer, except neurons in CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.
- However, the neurons in both layers still compute dot products, so their functional form is identical.
- Therefore, it is possible to convert between FC and CONV layers.

ConvNet Architectures

We have seen that Convolutional Networks are commonly made up of only three layer types:

- CONV
- POOL (assume max pooling)
- FC
- will add RELU activation function as a layer, which applies element-wise non-lineraity

To cut to the chase, 90% of applications involve selecting whatever architecture currently works best on ImageNet, download a pre-trained model and fine tune it on your data.

Case Studies

- ***LeNet***
  First successful application of Convolutional Networks
  Developed by Yann LeCun in 1990s
  Used to read zip codes, digits, etc.
- ***AlexNet***
  First work to popularise Convolutional Networks on Computer Vision
  Developed by Alex Krizhevsky, Ilya Sutskever & Geoff Hinton
  Won ImageNet ILSVRC challenge in 2012
  Similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other, rather than a single CONV layer immediately followed by a POOLING layer.
- ***ZF Net***
  Matthew Zeiler & Rob Fergus won 2013 ImageNet ILSVRC challenge with this architecture
  Improved upon AlexNet by tweaking architecture hyper-parameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.
- ***GoogLeNet***
  Szegedy et al from Google won 2014 ImageNet ILSVRC challenge with this architecture
  Main contribution was the development of an Inception Module, most recently Inception v4, which dramatically reduced the number of parameters in the network (4M, compared to AlexNet's 64M).
  Uses Average Pooling instead of fully connected layers at the top of the ConvNet, eliminating lots of parameters that don't seem to matter
- ***VGGNet***
  Karen Simonyan and Andrew Zisserman caaaame 2nd in 2014.
  Main contribution was in showing that the depth of the network is a critical component for good performance.
- ***ResNet***
  Kaiming et al won 2015 challenge
  Features special skipped connections, and a heavy use of batch normalisation.
  Also missing fully connected layers at the end of the network

State of the art in Convolutional Neural Network models and are the default choice for using ConvNets in practice.
Recent developments that tweak the original architecture can be found in Identity mappings in deep residual networks by Kaming et al.

2016: State of play between frameworks

Comparative Study of Deep Learning Software Frameworks

TensorFlow is new to the party!

Abstract

Deep learning methods have resulted in significant performance improvements in several application domains, namely computer vision, speech recognition, and natural language processing, and as such several software frameworks have been developed to facilitate their implementation. This paper presents a comparative study of five deep learning frameworks, namely Caffe, Neon, TensorFlow, Theano, and Torch, on three aspects: extensibility, hardware utilisation, and speed. The study is performed on several types of deep learning architectures and we evaluate the performace of the above frameworks when employed on a single machine for both (multi-threaded) CPU and GPU (Nvidia Tita X) settings. The speed performance metrics used here include the gradient computation time, which is important during the training phase of deep networks, and the forward time, which is importtant from the deployment perspective of trained networks. For convolutional networks, we also report how each of these frameworks support various convolutional algorithms and their corresponding performance. From our experiments, we observe that Theano and Torch are the most easily extensible frameworks. We observe that Torch is best suited for any deep architecture on CPU, followed by Theano. It also achieves the best performance on the GPU for large convolutional and fully connected networks, followed closely by Neon. Theano achieves the best performance on GPU for training and deployment of LSTM networks. Caffe is the easiest for evaluating the performance of standard deep architectures. Finally. TensorFlow is a very flexible framework, similar to Theano, but its performance is currently not competitive compared to the other studied frameworks.

Introduction

We evaluate these frameworks from the perspective of practioners, on the following aspects:

- Extensibility
  Their capability to incorporate different types of deep learning architectures (convolutional, fully-connected, and recurrent networks), different training procedures (unsupervised layer-wise re-training and supervised learning), and different convolutional algorithms (e.g. FFT-based algorithm).
- Hardware utilisation
  Their efficacy to incorporate hardware resources in either (multi-threaded) CPU or GPU setting.

- Speed
  Their speed performance from both training and deployement perspectives

Overview of the deep learning frameworks

See paper Appendix

Benchmarking Setup

Evaluation Metrics

We use the two following evaluation metrics to obtain a holistic understanding of speed of the five deep learning frameworks under various system scenarios and application domains:

- Forward Time
  Time it takes for an input batch of a pre-selected batch size, for a given dataset and network, to flow through the entire network and produce the corresponding output. This is important as it indicates the latency of a deep neywork when deployed in the real-world.
- Gradient Comutation Time
  Time it takes to get the gradients for each measuarable parameter in the deep network for a given input batch. This is an important indicator of the training time.

System Setup

All the experimets were run on a single machine – see paper for details.

Results and Discussions

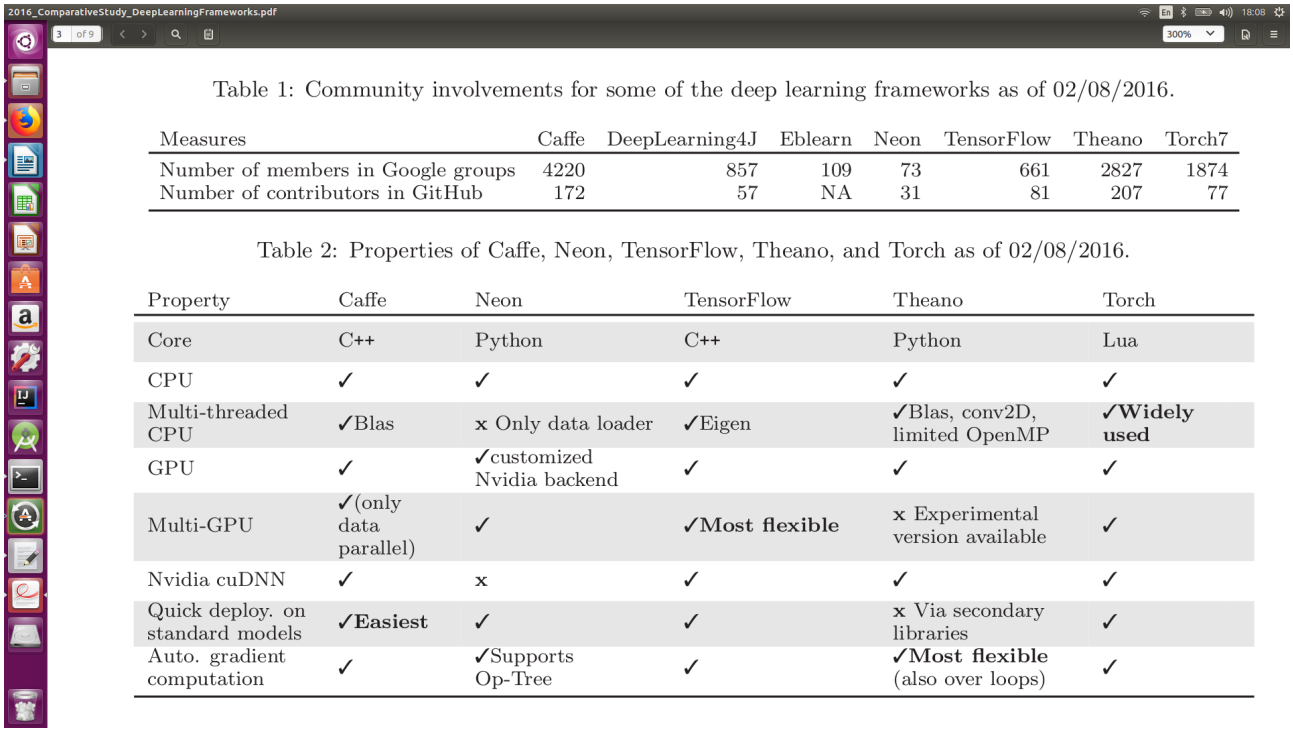


Table 1: Community involvements for some of the deep learning frameworks as of 02/08/2016.

Measures	Caffe	DeepLearning4J	Eblearn	Neon	TensorFlow	Theano	Torch7
Number of members in Google groups	4220	857	109	73	661	2827	1874
Number of contributors in GitHub	172	57	NA	31	81	207	77

Table 2: Properties of Caffe, Neon, TensorFlow, Theano, and Torch as of 02/08/2016.

Property	Caffe	Neon	TensorFlow	Theano	Torch
Core	C++	Python	C++	Python	Lua
CPU	✓	✓	✓	✓	✓
Multi-threaded CPU	✓Blas	x Only data loader	✓Eigen	✓Blas, conv2D, limited OpenMP	**✓Widely used**
GPU	✓	✓customized Nvidia backend	✓	✓	✓
Multi-GPU	✓(only data parallel)	✓	**✓Most flexible**	x Experimental version available	✓
Nvidia cuDNN	✓	x	✓	✓	✓
Quick deploy. on standard models	**✓Easiest**	✓	✓	x Via secondary libraries	✓
Auto. gradient computation	✓	✓Supports Op-Tree	✓	**✓Most flexible** (also over loops)	✓

Illustration 6: Table 1: Community Involvement & Table 2: Framework Properties

The evaluations are performed by training stacked autoencoders and convolutional networks on the MNIST and ImageNet datasets as well as training LSTM network using the IMDB review dataset.

LeNet

TensorFlow code samples: https://github.com/soumith/convnet-benchmarks

The first benchmark is a slightly modified LeNet neural network on the MNIST dataset where the sigmoid activations are replaced by ReLU units and softmax logistic layer is used instead of the RBF network. It consists of two convolutional-pooling layers with the tanh activation functions and two fully connected layers.

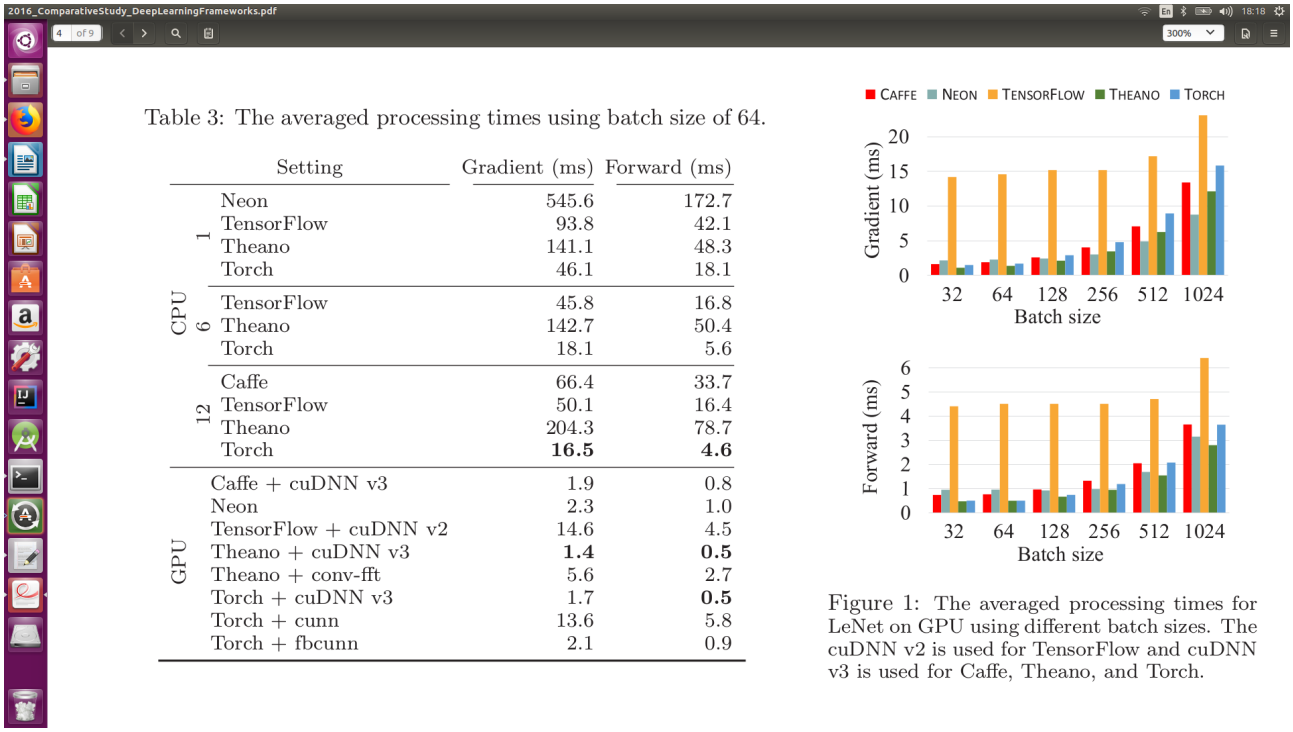


Table 3: The averaged processing times using batch size of 64.

		Setting	Gradient (ms)	Forward (ms)
CPU	1	Neon	545.6	172.7
		TensorFlow	93.8	42.1
		Theano	141.1	48.3
		Torch	46.1	18.1
	6	TensorFlow	45.8	16.8
		Theano	142.7	50.4
		Torch	18.1	5.6
	12	Caffe	66.4	33.7
		TensorFlow	50.1	16.4
		Theano	204.3	78.7
		Torch	**16.5**	**4.6**
GPU		Caffe + cuDNN v3	1.9	0.8
		Neon	2.3	1.0
		TensorFlow + cuDNN v2	14.6	4.5
		Theano + cuDNN v3	**1.4**	**0.5**
		Theano + conv-fft	5.6	2.7
		Torch + cuDNN v3	1.7	**0.5**
		Torch + cunn	13.6	5.8
		Torch + fbcunn	2.1	0.9

Figure 1: The averaged processing times for LeNet on GPU using different batch sizes. The cuDNN v2 is used for TensorFlow and cuDNN v3 is used for Caffe, Theano, and Torch.

Illustration 7: LeNet on MNIST Metrics

AlexNet

TensorFlow code samples: https://github.com/soumith/convnet-benchmarks

In this section, we train AlexNet (5 convolutional layers, 3 pooling layers, 2 fully connected layers with ReLU and drop out, and a softmax logistic loss) on the ImageNet dataset. Note that there have been many recent, larger networks like GoogLeNet, OxfordNet, etc. but we stick with AlexNet as it is the first network that significantly improved performance on ImageNet and is very popular.

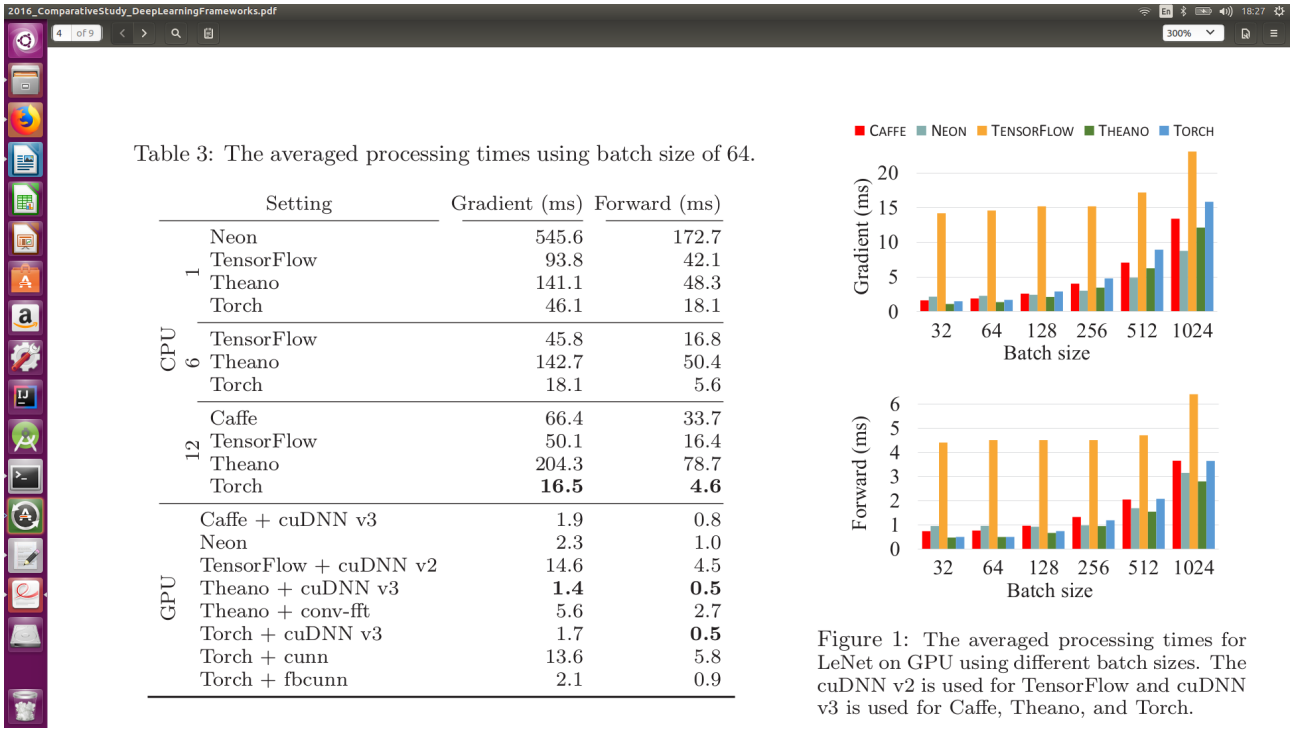


Table 3: The averaged processing times using batch size of 64.

		Setting	Gradient (ms)	Forward (ms)
CPU	1	Neon	545.6	172.7
		TensorFlow	93.8	42.1
		Theano	141.1	48.3
		Torch	46.1	18.1
	6	TensorFlow	45.8	16.8
		Theano	142.7	50.4
		Torch	18.1	5.6
	12	Caffe	66.4	33.7
		TensorFlow	50.1	16.4
		Theano	204.3	78.7
		Torch	**16.5**	**4.6**
GPU		Caffe + cuDNN v3	1.9	0.8
		Neon	2.3	1.0
		TensorFlow + cuDNN v2	14.6	4.5
		Theano + cuDNN v3	**1.4**	**0.5**
		Theano + conv-fft	5.6	2.7
		Torch + cuDNN v3	1.7	**0.5**
		Torch + cunn	13.6	5.8
		Torch + fbcunn	2.1	0.9

Figure 1: The averaged processing times for LeNet on GPU using different batch sizes. The cuDNN v2 is used for TensorFlow and cuDNN v3 is used for Caffe, Theano, and Torch.

Illustration 8: AlexNet on ImageNet Metrics

Stacked Autoencoders

LSTM

Sentiment analysis on the IMDB database.

Conclusions

From our experiments, we observe that Theano and Torch are the most easily extensible frameworks. We observe that Torch is best suited for any deep architecture on CPU, followed by Theano. It also achieves the best performance on the GPU for large convolutional and fully connected networks, followed closely by Neon. Theano achieves the best performance on GPU for training and deployment of LSTM networks. Caffe is the easiest for evaluating the performance of standard deep architectures. Finally. TensorFlow is a very flexible framework, similar to Theano, and specially in employing homgeneous/heterogeneous devices for the various parts of the computational graph, but its performance is currently not competitive compared to the other studied frameworks.

1998: LeNet: First success of CNNs for image recognition

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner

Abstract

Multilayer Neural Networks with the ***backpropagation algorithm*** constitute the best example of a successful ***Gradient-Based Learning*** technique. Given an appropiate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal preprocessing. This paper reviews various methods applied to handwritten character recognition and compares them on a standard handwritten digit recognition task. Convolutional Neural Networks, that are specifically designed to deal with the variability of 2D shapes, are shown to outperform all other techniques.

Real-life document recognition systems are composed of multiple modules including field extraction, segmentation, recognition, and language modeling. A new learning paradigm, called ***Graph Transformer Networks (GTN)***, allows such multi-module systems to be trained globally using Gradient-Based methods so as to minimise an overall performance measure.

Two systems for on-line handwriting recognition are described. Experiments demonstrate the advantage of global training, and the flexibility of Graph Transformer Networks.

A Graph Transformer Network for reading bank cheques is also described. It uses Convolutional Neural Network character recognisers combined with global training techniques to provide record accuracy on business and personal cheques. It is deployed commercially and reads several million cheques per day.

Introduction

The main message of this paper is that better pattern recognition systems can be built by relying more on automatic learning, and less on hand designed heuristics. This is made possible by recent progress in machine learning and computer technology. Using character recognition as a case study, we show that hand crafted feature extraction can be advantageously replaced by carefully designed learning machines that operate directly on pixel images. Using document understanding as a case study, we show that the traditional way of building recognition systems by manually integrating individually designed modules can be replaced by a unified and well-principled design paradigm, called Graph Transformer Networks, that allow training all the modules to optimise a global performance criterion.

A combination of three factors have changed the need for appropriate feature extractors:

- Availability of **low-cost machines** with fast arthimetic units
  rely on brute-force “numerical” methods than on algorithmic refinements
- Availability of **large databases**
  rely more on real data and less on had crafted feature extraction
- Availabilty of **powerful machine learning techniques**
  can handle high-dimensional inputs
  generate intricate decision functions when fed with these large data sets

Section II : CNNs for Isolated Character Recognition

Consider tasks of handwritten character recognition. While more automatic learning is beneficial, no learning technique can succeed without a minimal amount of prior knowledge about the task. In the case of multi-layer neural networks, ***a good way to incorporate knowledge is to tailor its architecture to the task***. Convolutional Neural Networks introduced in Section II are an example of specialised neural network architectures which incorporate knowledge about the invariances of 2D shapes by using local connection patterns, and by imposing constraints on the weights.

Section III: Results & Comparison with other methods

Compare the performance of several learning techniques on a benchmark data set for handwritten digit recognition.

Section IV: multi-module systems & Graph Transformer Networks (GTN)

To go from recognition of individual characters to the recognition of words and sentences in documents, the idea of combining multiple modules trained to reduce the overall eror is introduced in Section IV. Recognising variable-length objects such as handwritten words using multi-module systems is best done if the modules manipulate directed graphs. This leads to the concept of trainable Graph Transformer Network (GTN) also introduced here.

Section V: Multiple object recognition: Heuristic over-segmentation

A description of the now classical method of heuristic over-segmentation for recognising words or other character strings.

Section VI: Global training for GTNs

Here discriminative and non-discriminative gradient-based techniques for training a recogniser at the word level without requiring manual segmentation and labelling are presented.

Section VII: Multiple object recognition: Space Displacement NN

Presents the promising Space-Displacement Neural Network approach that eliminates the need for segmentation heuristics by scanning a recogniser at all possible locations on the input.

Section VIII: Graph Transformer Networks & Transducers

It is shown that Graph Transformer Networks can be formulated as multiple generalised transductions, based on a general graph composition algorithm. The connections between GTNs and Hidden Markov Models, commonly used in speech recognition is also treated.

Section IX: An on-line handwriting recognition system

Describes a globally trained GTN system for recognising handwritting entered in a pen computer. The results clearly demonstrate the advantages of training it on pre-segmented, hand-labeled, isolated characters.

Section X: A Cheque reading system

Descibes a compete GTN-based system for reading handwritten and machine-printed bank cheques. The core of the system is the ***LeNet-5*** described in Section II.

Section 1:

Learning from Data

One of the most successful approaches to automatic machine learning can be called “numerical” or ***gradient-based learning***. The learning machine computes a function:

$$Y^p = F(Z^p, W) \textbf{ \# forward pass}$$

where Z^p is the p-th input pattern, and W represents the collection of adjustable parameters in the system. In a pattern recognition setting, the output Y^p may be interpreted as the recognised class label of pattern Z^p, or as scores or probabilities associated with each class. A ***loss function***:

$$E^p = D(D^p , F(W, Z^p)) \textbf{ \# loss function}$$

measures the discrepancy between D^p, the “correct” or desired output, i.e. label, for pattern Z^p, and the output produced by the system. The ***average loss function*** $E_{train}(W)$ is the average of the errors E^p over a set of labeled examples called the training set $\{(Z^1, D^1), \ldots.(Z^p, D^p)\}$.

In the simplest setting, the learning problem consists in finding the value W that ***minimises*** $E_{train}(W)$. In practice, the performance is estimated by measuring the accuracy on a set of samples disjoint from the training set, called the test set. The gap between the expected error rate on the test set E_{test} and the error rate on the training set E_{train} decreases with the number of training samples.

Gradient-Based Learning

Gradient-Based Learning draws on the fact that it is generally much easier to minimise a resonably smooth, continuous function than a discrete (combinatorial) function. The loss function can be minimised by estimating the ***impact of small variations of the parameter values on the loss function***. This is measured by the gradient of the loss function w.r.t. the parameters (***partial derivatives***). Efficient learning algorithms can be devised when the gradient vector can be computed ***analytically*** (as opposed to numerically through perturbations). This is the basis of numerous gradient-based learning algorithms with continuous-valued parameters. In the procedures described

here, the set of parameters W is a real-valued vector, w.r.t. E(W) which is continuous, as well as differentiable almost everywhere.

The ***simplest minimisation procedure*** in a such a setting is the ***gradient descent algorithm*** where W is iteratively adjusted as follows:

$$W_k = W_{k-1} - \varepsilon \, (\partial E(W) / \partial W) \; \textbf{\textit{\# update network weights}}$$

In the simplest case, ε is a scalar constant. More sophisticated procedures use variable ε, although their usefulness to large learning machines is very limited.

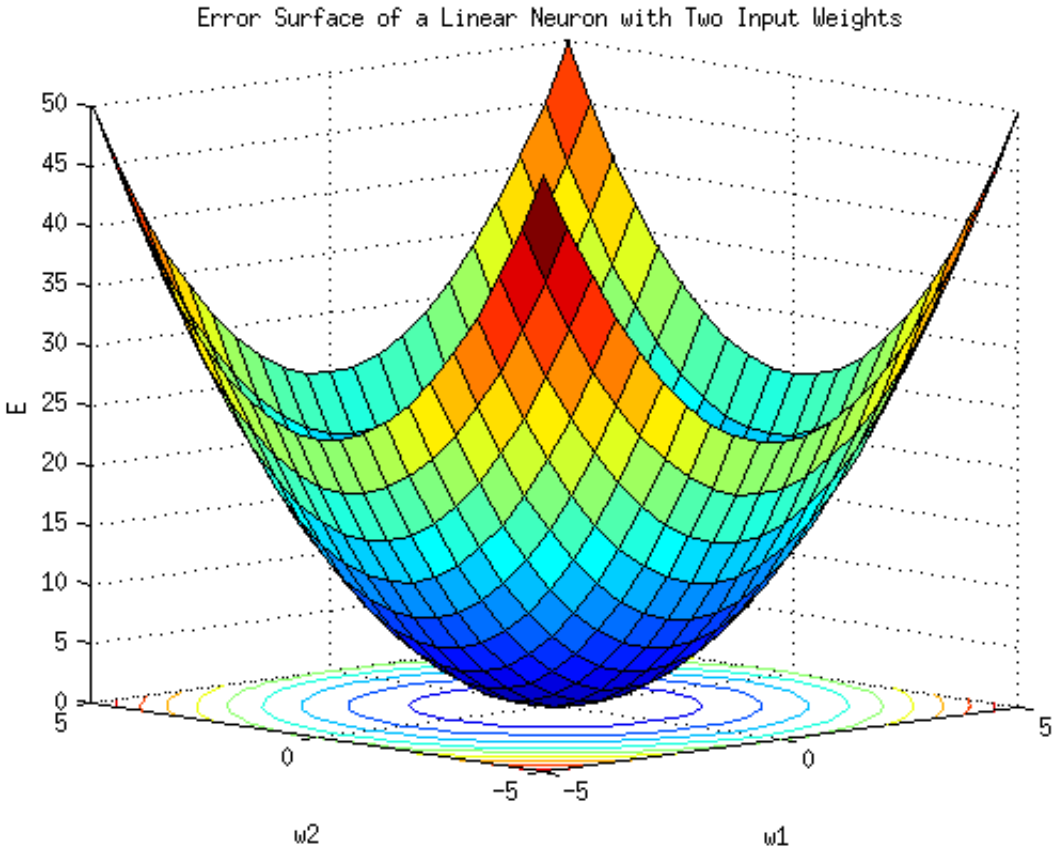


Illustration 9: Error Surface of a Linear Neuron with 2 Input Weights, w1 & w2

A popular minimisation procedure is the ***stochastic gradient algorithm***, also called the on-line update. It consists in updating the parameter vector using a noisy, or approximated, version of the average gradient. In the most common instance of it, W is updated on the basis of a single sample:

$$W_k = W_{k-1} - \varepsilon \, (\partial E^{pk}(W) / \partial W) \; \textbf{\textit{\# update network weights}}$$

With this procedure the parameter vector fluctuates around an average trajectory, but usually converges considerably faster than regular gradient descent on large training sets with redundant samples (such as those found in speech or character redognition).

We now need to compute the gradient, $\partial E^{pk}(W) / \partial W$, for that we use backward propagation.

Gradient Back-Propagation

The basic idea of back-propagation is that gradients can be computed efficiently ***by propagation from the output to the input***. As an aside, it appears odd that ***local minima*** do not seem to be a problem for multi-layer neural networks is somewhat of a theoretical mystery, but it appears that a network that is over-sized for a task, the presence of extra dimensions in parameter space ***reduces the risk of unattainable regions***.

Back-propagation is by far the most widely used neural-network learning algorithm, and probably the most widely used learning algorithm of any form.

The back-propagation procedure is used to compute the gradients of the loss function w.r.t. all the parameters in the system. For example, let us consider a system built as a cascade of modules, each of which implements a function:

$$X_n = F_n(W_n, X_{n-1}) \textbf{ \# layer computation}$$

where:

- X_n is a vector representing the output of the module,
- W_n is the vector of tunable parameters in the module (a subset of W),
- and X_{n-1} is the module's input vector (as well as the previous module's output vector).

The input X_0 to the first module is the input pattern Z^p. If the partial derivative of E^p w.r.t. X_n is known, then the partial derivatives of E^p w.r.t. W_n and X_{n-1} can be computed using the backward recurrence:

$$\partial E^p / \partial W_n = \partial F / \partial W (W_n, X_{n-1}) \partial E^p / \partial X_n \text{ \# } \textbf{backward propagation}$$

$$\partial E^p / \partial X_{n-1} = \partial F / \partial X (W_n, X_{n-1}) \partial E^p / \partial X_n \text{ \# backward propagation}$$

where:

- $\partial F / \partial W (W_n, X_{n-1})$ is the Jacobian of F w.r.t. W evaluated at the point (W_n, X_{n-1}) and
- $\partial F / \partial X (W_n, X_{n-1})$ is the Jacobian of F w.r.t. X.

The Jacobian of a vector function is a matrix contining the partial derivatives of all the ouputs w.r.t. all the inputs.

The first equation computes some terms of the gradient of $E^p(W)$, while the second equation generates a backward recurrence, as in the well-known back-propagation procedure for neural networks. We can average the gradients over the training patterns to obtain the full gradient.

Traditional multi-layer neural networks are a special case of the above where the state information X_n is represented with fixed-sized vectors, and where the modules are alternated layers of matrix multiplications (the weights) and component-wise sigmoid functions (the neurons).

However, the state information in compex recognition systems is best represented by ***graphs*** with numerical information attached to the arcs. In this case, each module, called a Graph Transformer,

takes one or more graphs as input, and produces a graph as output. This means that Gradient-Based Learning can be used to train all the parameters in all the modules so as to ***minimise a global loss function***.

Pseudocode for a stochastic gradient descent optimisation algorithm:

initialise ***model*** weights (often small random values)

do

forEach training example named ex

prediction = neural_net_output(network, ex) # ***forward pass***

actual = label(ex) # ***label***

compute error (prediction – actual) # ***loss function***

compute gradients, $\partial E^{pn}(W) / \partial W$ # ***backward propagation***

$W_n = W_{n-1} - \varepsilon (\partial E^{pn}(W) / \partial W)$ # ***update network weights***

until all examples classified properly or other stopping criterion satisfied

return trained model

Equivalent TensorFlow code to train LeNet

A High-Level API implementation of MNIST.py can be found at:

$ cd ~/ResearchProject/ImageTestBeg/Applications/mnist/
$ (tensorflow) python mnist.py # to train model
$ tensorboard –logdir=./mnist_model # explore pre-baked model

 # Loss function
loss = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)

 # Configure the Training Op (Optimiser)
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
train_op = optimizer.minimize(
loss=loss,
global_step=tf.train.get_global_step())

 # Create the Estimator (TensorFlow High Level API)
cnn_model_fn = tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op) ***# model***
mnist_classifier = tf.estimator.Estimator(model_fn=cnn_model_fn) ***# classifier***

 # Train the model
train_input_fn = tf.estimator.inputs.numpy_input_fn(
x={"x": train_data}, y=train_labels,
batch_size=100,

```
        num_epochs=None,
        shuffle=True)

mnist_classifier.train(
        input_fn=train_input_fn,
        steps=20000)
```

Section II

The ability of multi-layer networks trained with gradient descent to learn complex, high-dimensional, non-linear mappings from large collections of examples makes them obvious candidates for image recognition tasks.

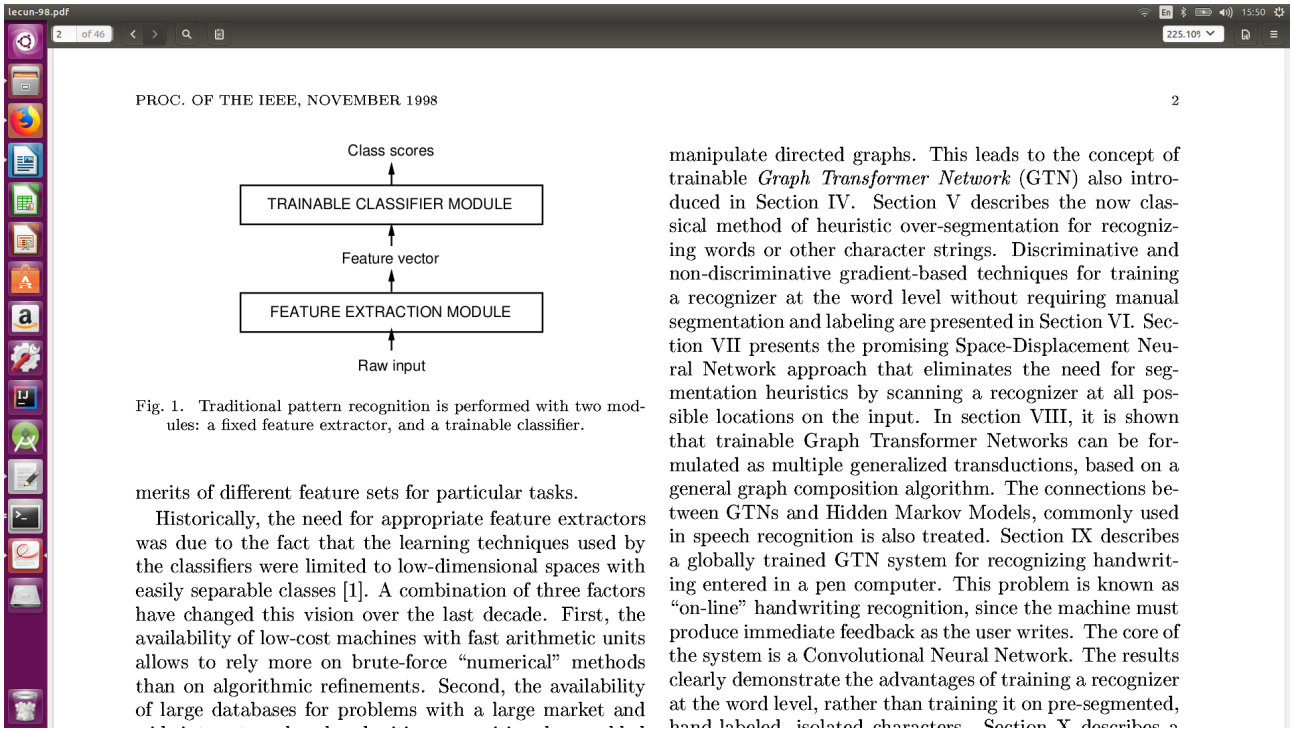


Illustration 10: Traditional Pattern Recognition Flow

The idea is to rely on as much as possible on learning the feature extractor itself, but as there are problems with fully connected layers, we need to structure the CNN with respect to the problem in hand.

There are a number of problems with fully connected layers for image processing:

- Typical ***images are large*** with several hunderd variables (pixels)
  This leads to a very large number of weights leading to a larger training set
- No built-in ***invariance*** with respect to translations, or local distortions of the inputs
  Meaning image must be approximately size-normalised & centered in the input field
- Topology of the input is entirely ignored
  This means that the input variables can be presented in any order w/o affecting training

But, images (or time-frequenct representations of speech) have a strong 2D structure, i.e. pixels that are spatially ot temporally nearby are highly correlated

In CNNs, ***shift invariance*** is automatically obtained by forcing the replication of weight configurations across space and CNNs force the ***extraction of local features*** by restricting the receptive fields of hidden units to be local.

Convolutional Networks

(Covered in Convolutional Neural Networks document)

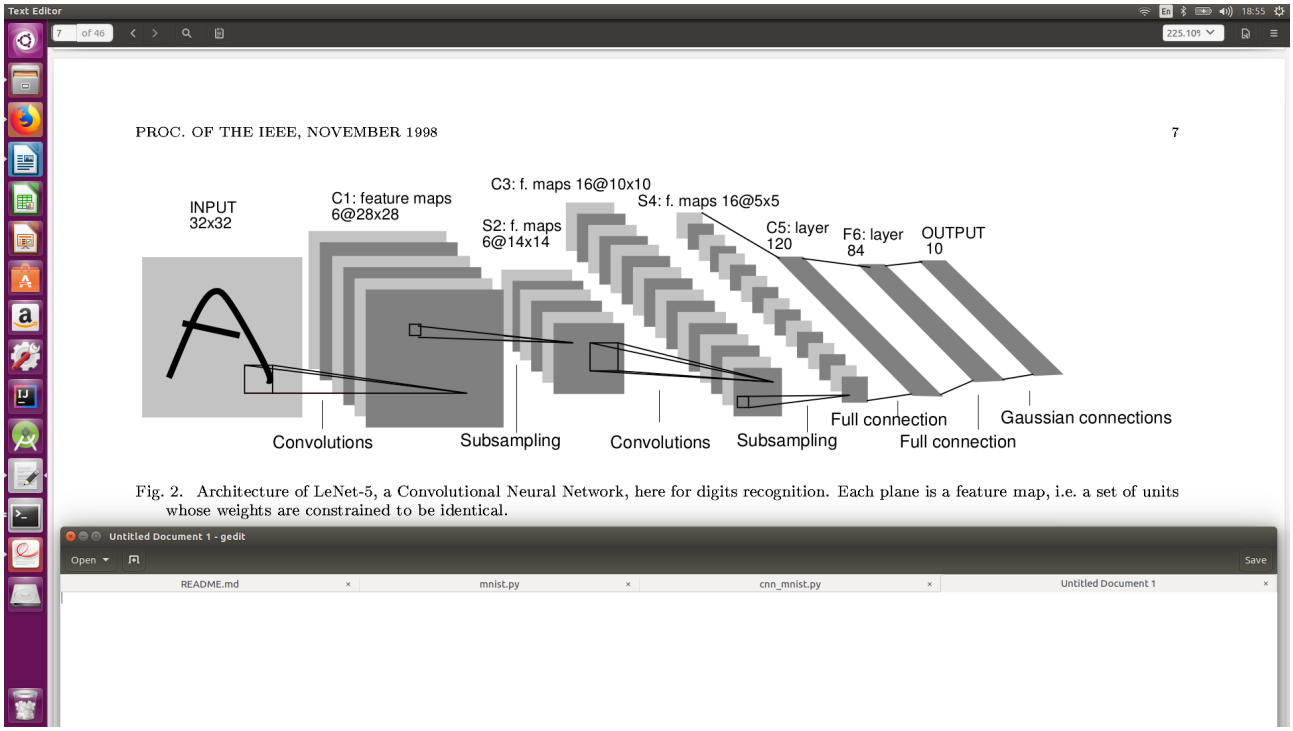


Illustration 11: LeNet-5 Architecture

Convolutional Networks combine three architectural ideas to ensure some degree of shift, scale, and distoration invariance:

- **Local Receptive Fields**
  Each unit in a layer receives inputs from a set of units located in a small neighbourhood in the previous layer. With local receptive fields, neurons can extract elementary visual features such as oriented edges, end-points, corners (or similar features in other signals such as speech spectrograms). These features are then combined by the subsequent layers in order to detect higher-order features.
  Input Plane receives images of characters that are approximately size normalised & centered
- **Shared Weights (or weight replication)**
  elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image. This knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors. The set of outputs of the units in such a plane is called a ***feature map***

Units in a feature map are all constrained to perform the same operation on different parts of the image, so multiple features can be extracted at each location

- **Spatial or temporal sub-sampling**
  Reduces the resolution of the feature map and reduces the sensitivity of the output to shifts and distortions

LeNet-5

LeNet-5 comprises 7 layers, not counting the input, all of which contain trainable parameters (weights). The values of the input pixels are normalised so that the background level (white) corresponds to a value of -0.1 and the foreground (black) corresponds to 1.175. This makes the mean input roughly 0, and the variance roughly 1 which accelerates learning.

In the following, covolutional layers are labelled Cx, sub-sampling layers are labeled Sx, and fully-connected layers are labelled Fx, where x is the layer index.

Each input image is of size 32x32.

Layer C1:
Convolutional layer with 6 feature maps. Each unit in each feature map is connected to a 5x5 neighbourhood in the input. The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. C1 contains 156 trainable parameters, and 122,304 connections.

Layer S2:
A sub-sampling layer with 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 neighbourhood in the corresponding feature map in C1. The four inputs to a unit in S2 are added, then multiplied by a trainable coefficient, and added to a trainable bias. The result is passed through a sigmoidal (ReLU fuction more recently) function. The 2x2 receptive fields are non-overlapping, therefore feature maps in S2 have half the number of rows and columns as feature maps in C1. Layer S2 has 12 trainable parameters and 5,880 connections.

Layer C3:
A convolutional layer with 16 feature maps. Each unit in each feature map is connected to several 5x5 neighbourhoods at identical locations in a subset of S2's feature maps.

Layer S4:
A sub-sampling layer with 16 feature maps of size 5x5. Each unit in each feature map is connected to a 2x2 neighbourhood in the corresponding feature map in C3, in a similar way as C1 and S2. Layer S4 has 32 trainable parameters and 2,000 connections.

Layer C5:
A convolutional layer with 120 feature maps. Each unit is connected to a 5x5 neighbourhood on all 16 of s4's feature maps. Here, because the size of s4 is also 5x5, the sixe of C5's feature map is 1x1: this amounts to a full connection between S4 and C5. C5 is has 48,120 trainable connections.

Layer F6:
Contains 84 units and is fully connected to C5. It has 10,164 trainable parameters.

Output layer:
is composed of Euclidean Radial Basis Function (RBF, but more recently Softmax is used), one for each class, with 84 inputs each.

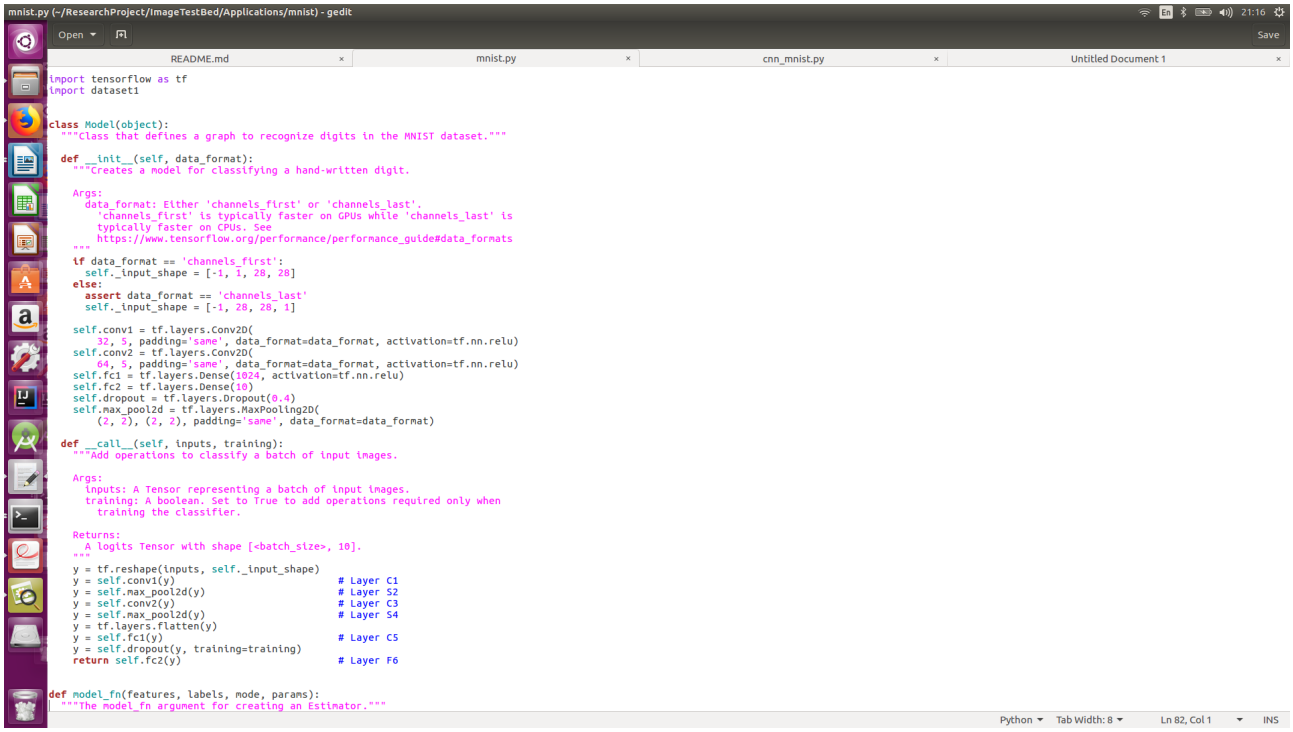


Illustration 12: LeNet expressed in TensorFlow High Level API Code

Section III: Results & Comparison with other methods

Though many existing methods combine a hand-crafted feature extractor and a trainable classifier, this study concentrates on adaptive methods that operate directly on size-normalised images.

When LeNet was run using Tensorflow on my laptop with the MNIST database as input, the test error rate was 0.73%, very similar to the 0.8% error rate recorded by LeCun et al.

TensorBoard

TensorFlow provides a utility called TensorBoard. One of TensorBoard's many capabilities is visualising a computation graph. This is easily done with a few simple commands:

```
writer = tf.summary.FileWriter('.')
writer.add_graph(tf.get_default_graph())
```

This will produce an event file in the current directory with a name in the following fomant:

`events.out.tfevents.{timestamp}.{hostname}`

In a general terminal, launch tensorBoard with:

`tensorboard --logdir .`

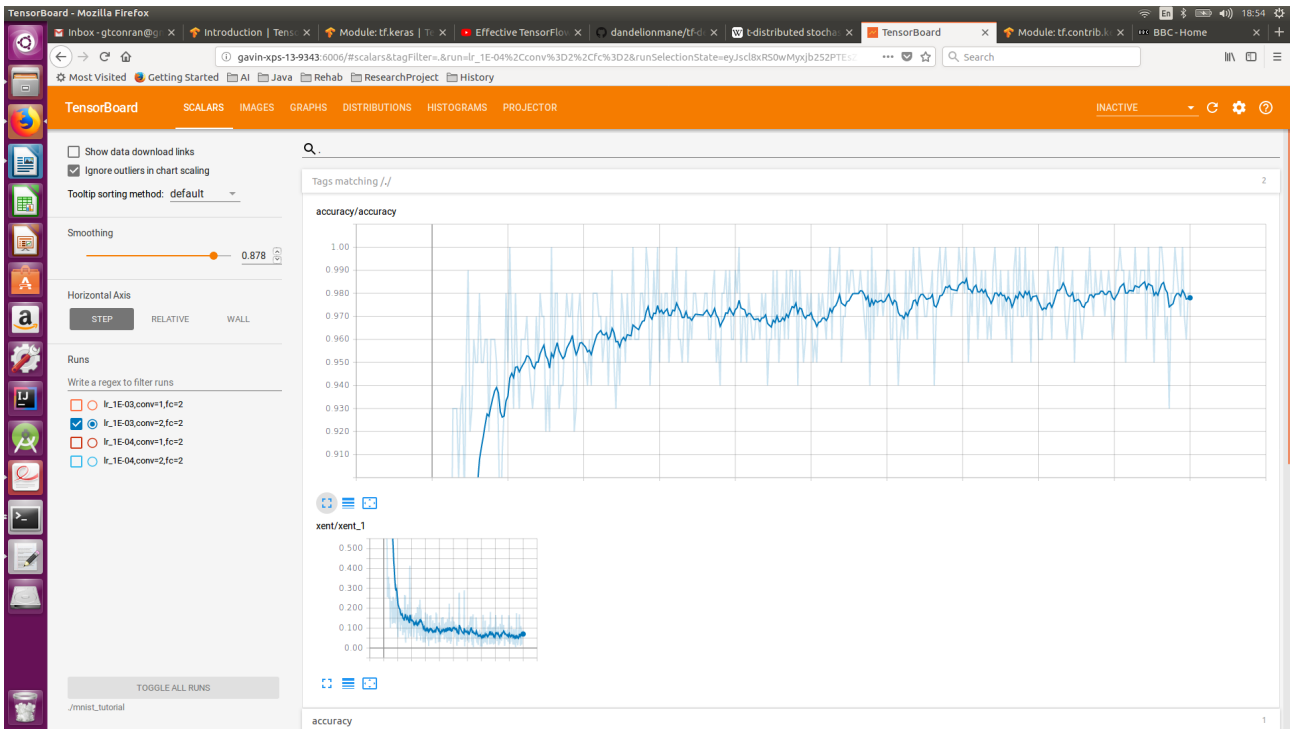


Illustration 13: LeNet Metrics Visualised by TensorBoard

Database: the Modified NIST set

The database used to train and test the systems described in this paper was constructed from the NIST special Database 3 and Special Database 1 containing binary images of handwritten digits.

Results

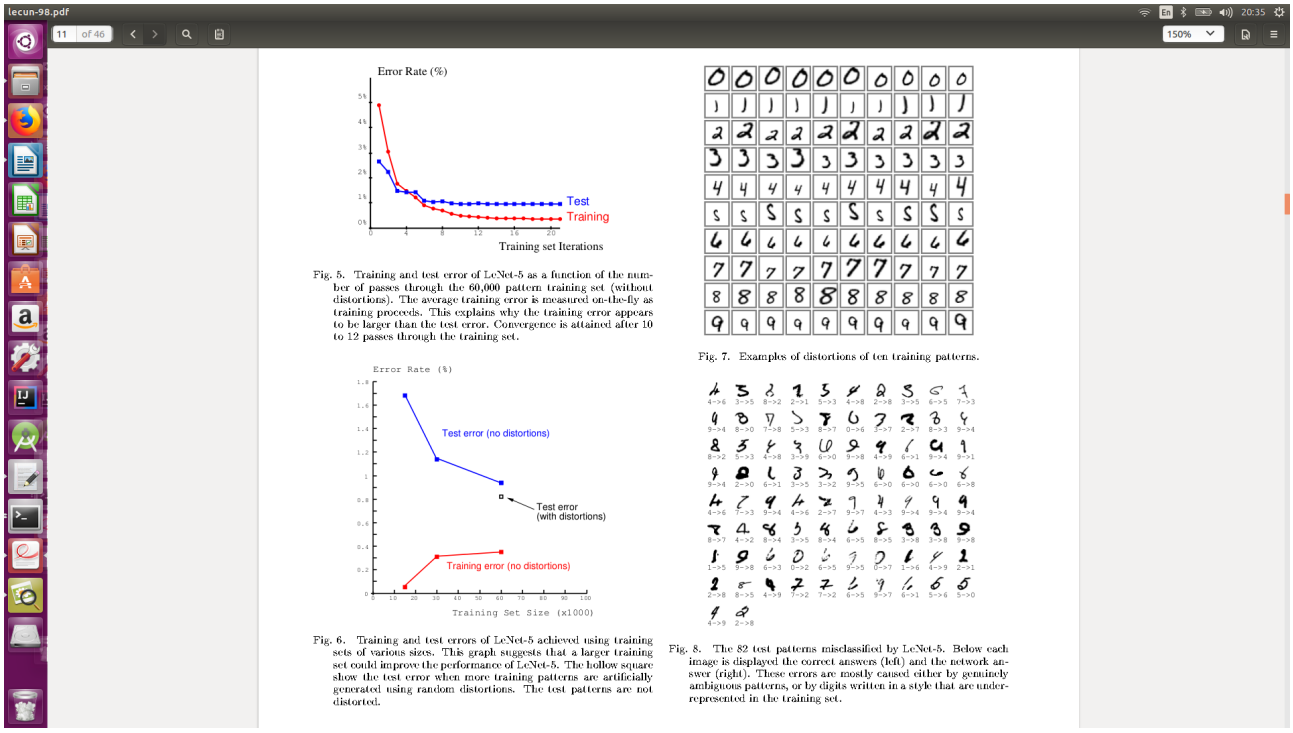


Fig. 5. Training and test error of LeNet-5 as a function of the number of passes through the 60,000 pattern training set (without distortions). The average training error is measured on-the-fly as training proceeds. This explains why the training error appears to be larger than the test error. Convergence is attained after 10 to 12 passes through the training set.

Fig. 6. Training and test errors of LeNet-5 achieved using training sets of various sizes. This graph suggests that a larger training set could improve the performance of LeNet-5. The hollow square show the test error when more training patterns are artificially generated using random distortions. The test patterns are not distorted.

Fig. 7. Examples of distortions of ten training patterns.

Fig. 8. The 82 test patterns misclassified by LeNet-5. Below each image is displayed the correct answers (left) and the network answer (right). These errors are mostly caused either by genuinely ambiguous patterns, or by digits written in a style that are underrepresented in the training set.

Illustration 14: LeNet-5 Results

Comparison with other classifiers

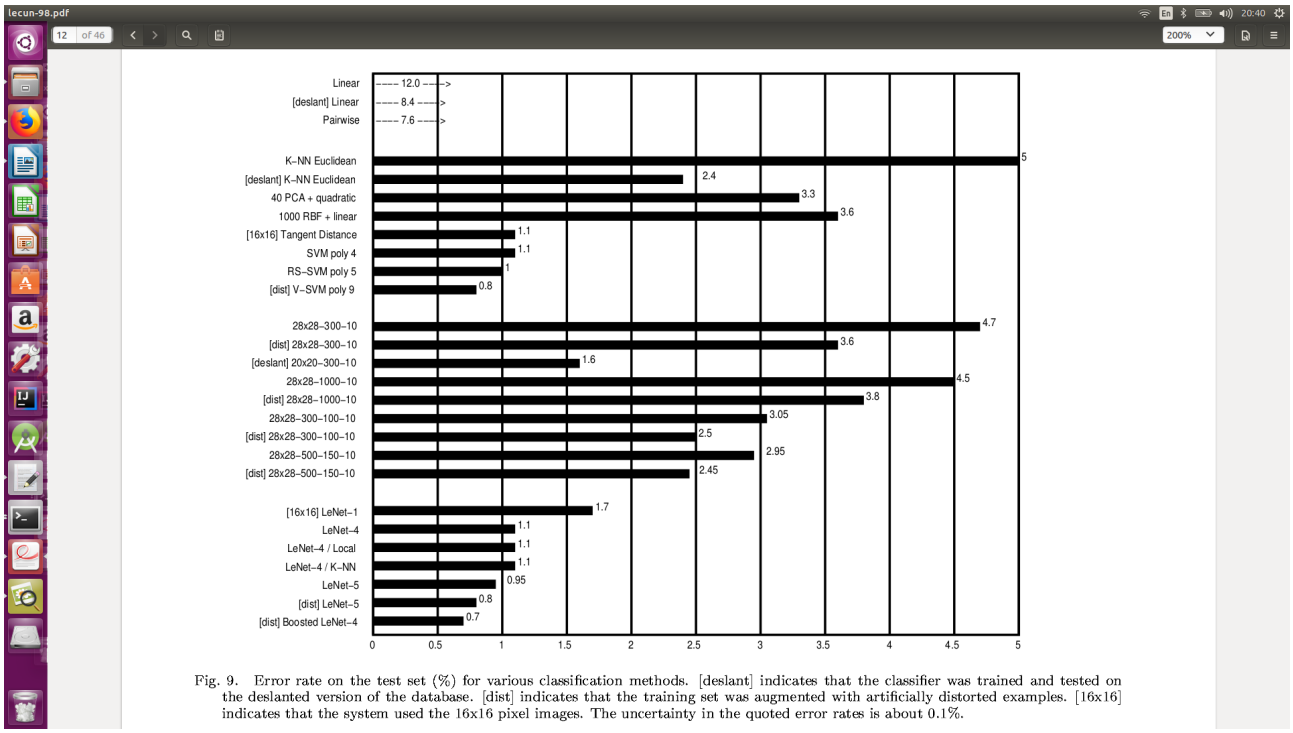


Fig. 9. Error rate on the test set (%) for various classification methods. [deslant] indicates that the classifier was trained and tested on the deslanted version of the database. [dist] indicates that the training set was augmented with artificially distorted examples. [16x16] indicates that the system used the 16x16 pixel images. The uncertainty in the quoted error rates is about 0.1%.

Illustration 15: Comparison with other classifiers

When plenty of data is available, many methods can attain respectable accuracy. The neural-net methods run much faster and require much less space than memory based techniques. The neural

nets' advantage will become more striking as training databases continue to increase in size → ImageNet!!!

Section IV: Multi-Module Systems & Graph Transformer Networks

The classical back-propagation algorithm is a simple form of Gradient-Based Learning. However, it is clear that the gradient back-propagation algorithm described earlier describes a more general situation than simple multi-layer feed-forward networks composed of alternated linear transformations and sigmoidal functions. In principle, derivatives can be back-propagated through any arrangement of functional modules, as long as we can compute the product of the Jacobians of those modules by any vector.

Why would we want to train systems composed of multiple heterogeneous modules?

The answer is that large and complex trainable systems need to be built out of simple, specialised modules. The simplest example is LeNet-5, whih mixes convolutional layers, sub-sampling, fully-connected layers, and RBF (Softmax) layers.

A less trivial example is a system for recognising words, that can be trained to simultaneously segment and recognise words, without ever being given the correct segmentation, e.g. Tesseract.

A multi-modal system is defined by the function implemented by each of the modules to each other. The graph implicitly defines a partial order according to which the modules must be updated in the forward pass. Modules may or may not have trainable parameters. Loss functions, which measure the performance of the system are usually the last module in the system. In this framework, there is no qualitative difference between trainable parameters (weights), external inputs and outputs (images, probabilities) and intermediate state variables (x).

An Object-Oriented Approach

Object-Oriented programming offers a particularly convenient way of implementing multi-module systems. Each module is an instance of a class. Module classes have a "forward propagation" method called fprop whose arguments are the inputs and outputs of the module. Complex modules can be constructed from simpler modules by simply defining a new class whose slots will contain the member modules and the intermediate state variables between those modules. The fprop method of the class simply calls the fprop methods of the member modules, with the appropriate intermediate state variables or external input and output as arguments.

We will limit the discussion to the case of ***Directed Acyclic Graphs, DAG***s, or feed-forward networks.

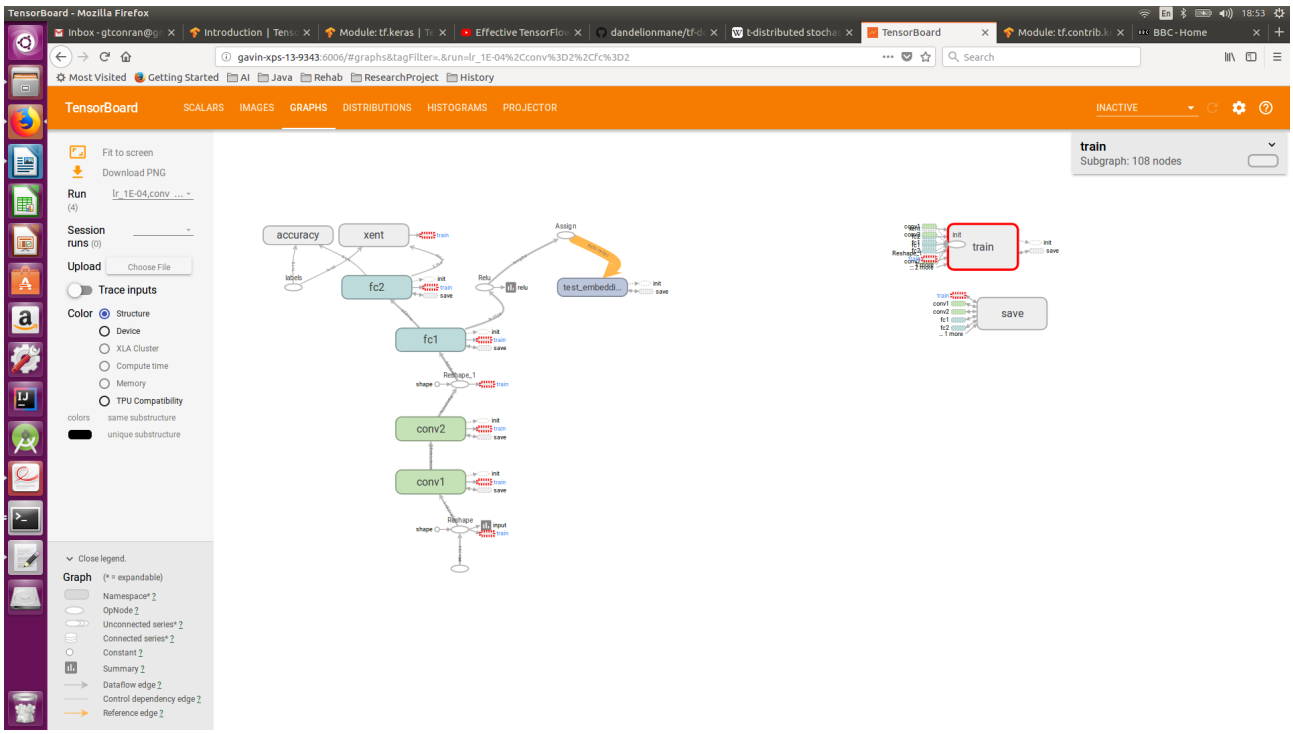


Illustration 16: LeNet Graph (DAG) Visualised by TensorBoard

Computing derivatives in a multi-module system is just as simple. A “backward propagation” method, called bprop, for each module class can be defined for that purpose. The bprop method of a module takes the same arguments as the fprop method. All the derivatives in the system can be computed by calling the bprop method on all the modules in reverse order compared to the forward propagation phase. The state variables are assumed to contain slots for storing the gradients computed during the backward pass, in addition to storage for the states computed in the forward pass. The backward pass effectively computes the partial derivatives of the loss E w.r.t. all the state variables (**X**) and all the parameters (**W**) in the system.

TensorFlow Low-Level APIs

A Low-Level API implementation of MNIST.py can be found at:

```
$ cd ~/ResearchProject/ImageTestBeg/Applications/mnist/tf-dev-summit-tensorboard-tutorial
$ (tensorflow) python mnist.py                    # train model(s)
$ tensorboard –logdir=./mnist_tutorial            # explore pre-baked model(s)
```

Tensor Values

The central unit of data in TensorFlow is the tensor. A tensor consists of a set of primitive values shaped into an array of any number of dimensions. A tensor's rank is its number of dimensions, while its shape is a tuple of integers specfying the array's length along each dimension.

TensorFlow uses numpy arrays to represent tensor **values**.

TensorFlow Core Walkthrough

You might think of TensorFlow Core programs as consisting of two discrete sections:

1. Building the computational graph (a tf.Graph)
2. Running the computational graph (using a tf.Session)

Graph

A computational graph is a series of tensorFlow operations arranged into a graph. The graph is composed into two types of objects:

- **Operations (or "ops")**
  The nodes of the graph.
  Operations describe calculations that consume and produce tensors
- **Tensors**
  The edges in the graph.
  These represent the values that will flow through the graph.
  Most TensorFlow functions return tt.Tensors

Each operation in a graph is given a unique name. This name is independent of the names the objects are assigned to in Python. Tensors are names after the operation that produces them followed bu an output index, e.g. "add:0".

Session

To evaluate tensors, instantiate a tf.Session object, informally known as a session. A session encapsulates the state of the TensorFlow runtime, and runs TensorFlow operations. If a tf.Graph is like a .py, a tf.Session is like the python executable.

When you request the output of a node with Session.run tensorFlow backtracks through the graph and runs all the nodes that provide input to the requested output node.

Feeding

A graph can be parameterised to accept external inputs, known as placeholders. A placeholder is a promise to provide a value later, like a function argument.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y
```

The preceding three lines are a bit like a function in which we define input parameters (x and y) and then an operation on them. We can evaluate this graph with multiple inputs by using the feed_dict argument of the run method to feed concrete values to the placeholders.

```
print(sess.run(z, feed_dict={x: 3, y: 4.5}))
print(sess.run(z, feed_dict={x: [1, 3], y: [2, 4]}))
```

This results in the following output:

```
7.5
[ 3.  7.]
```

Datasets

Placeholders work for simple experiments, but Datasets are the preferred method of streaming data into a model.

To get a runnable tf.Tensor from a Dataset you must first convert it to a tf.data.Iterator, and then call the Iterator's get_next method.

The simplest way to create an Iterator is with the make_one_shot_iterator method.

Foe more details see: https://www.tensorflow.org/programmers_guide/datasets

Layers

A trainable model must modify the values in the graph to get new outputs with the same input. Layers are the preferred way to add trainable parameters to a graph.

Layers package together both the variables and the operations that act on them, sounds like a ***class*** to me. For example, a densely-connected layer performs a weighted sum across all inputs for each output and applies an optional activation function. The connection weights and biases are managed by the layer object.

Creating, Initialising & Executing Layers

The following code creates a Dense layer that takes a batch of input vectors, and produces a single output value for each. To apply a layer to an input, call the layer as if it were a function. For example:

```
x = tf.placeholder(tf.float32, shape=[None, 3])
linear_model = tf.layers.Dense(units=1)
y = linear_model(x)
```

The layer contains variables that must be initialised before they can be used. While it is possible to initialise variables individually, you can easily initialise all the variables in a TensorFlow grapgh as follows:

```
init = tf.global_variables_initializer()
sess.run(init)
```

We can evaluate the linear-model's output tensor as we would any other tensor:

```
print(sess.run(y, {x: [[1, 2, 3],[4, 5, 6]]}))
```

will genarate a tw-element output vector such as the following:

```
[[-3.41378999]
 [-9.14999008]]
```

Layer Function shortcuts

For each layer class (like tf.layers.Dense) TensorFlow also supplies a shortcut function (like tf.layers.dense. The only difference is that a short cut function will create and run the layer in a single call. For example, the folowing code is equivalent to the earlier version:

```
x = tf.placeholder(tf.float32, shape=[None, 3])
y = tf.layers.dense(x, units=1)
init = tf.global_variables_initializer()
sess.run(init)
print(sess.run(y, {x: [[1, 2, 3], [4, 5, 6]]}))
```

While convenient, this approach allows no access to the tf.layers.Layer object, making introspection and debugging difficult, and reuse impossible.

Feature Columns

The easiest way to experiment with feature columns is using the tf.feature_column.input_layer function. This function only accepts dense columns as inputs, so to view the result of a categorical column you must wrap it in a tf.feature_column.indicator_column.

Training

Let's train a small regression model manually.

```
# Define the data
x = tf.constant([[1], [2], [3], [4]], dtype=tf.float32)
y_true = tf.constant([[0], [-1], [-2], [-3]], dtype=tf.float32)

# Define the model
linear_model = tf.layers.Dense(units=1)
y_pred = linear_model(x)

# Loss
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y_pred)

# Training
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

# Initialise the model
```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Run the model
for i in range(100):
  _, loss_value = sess.run((train, loss))
  print(loss_value)
print(sess.run(y_pred))
```

With the Low-Level APIs, all the differentiating mathematics of back propagation and weight updates have been abstracted away but you get the tf.Graph and tf.Session commands so you get the notion of executing through a graph. Graphs and Sessions are abstracted away with the High-Level APIs.

Chapter 2: How the backpropagation algorithm works

Neural Networks & Deep Learning by Michael Neilsen

http://neuralnetworksanddeeplearning.com

The four fundamental equations behind back-propagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives:

$$\partial C / \partial w^l_{jk} \text{ and } \partial C / \partial b^l_j$$

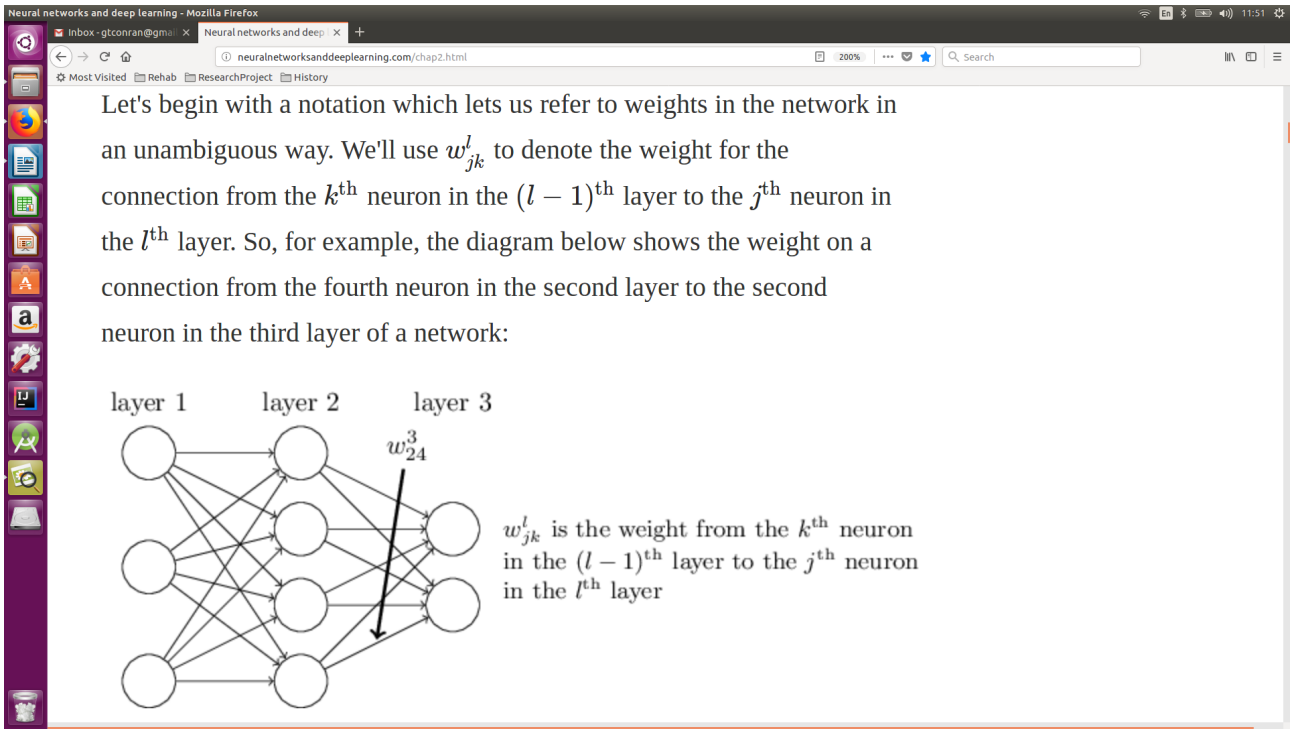


Illustration 17: Neural Network Notation

Computing the deltas

But to compute those, we first introduce an intermediate quantity, $\boldsymbol{\delta^l_j}$, which we call the ***error*** (or delta) in the j-th neuron in the l-th layer. Back-propagation will give us a procedure to compute the error, δ^l_j, and then will relate $\boldsymbol{\delta^l_j}$ to $\partial C / \partial w^l_{jk}$ and $\partial C / \partial b^l_j$. This is referred to as "computing the deltas".

We define the error, δ^l_j, of a neuron j in layer l by:

$$\delta^l_j \equiv \partial C / \partial z^l_j$$

We use δ^l to denote the vector of errors associated with layer l. Backpropagation will give us a way of computing δ^l for every layer, and then relating those errors to the qualities of real interest, $\partial C/\partial w^l_{jk}$ and $\partial C/\partial b^l_j$.

Four fundamental equations behind back-propagation

Backpropagation is based around four fundamental equations. Together, those equations give us a way of computing both the error, δ^l, and the gradient of the cost function.

BP-1: An equation for the error in the OUTPUT layer, δ^L

The components of δ^L are given by:

$$\delta^L_j = \delta C/\delta a^L_j \cdot \sigma'(z^L_j) \qquad \text{(BP-1)}$$

where

- $\delta C/\delta a^L_j$ measures how fast the cost function is changing wrt the j-th output activation
- $\sigma'(z^L_j)$ measures how fast the activation function, σ, is changing at z^L_j
- $z^L = w^L \cdot a^{L-1} + b^L$
- $a^L = \sigma(z^L)$

Rewritten in matrix form:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

where:

- $\nabla_a C$ is a vector of the partial derivatives $\delta C/\delta a^L_j$

As an example, in the case of the quadratic cost we have $\nabla_a C = (a^L - y)$, and so the fully matrix-based form of BP-1 becomes:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)$$

BP-2: An equation for the error δ^l in terms of the eror in the NEXT layer δ^{l+1}

In particular:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP-2)}$$

Where:

- $(w^{l+1})^T$ is the transpose of the weight w^{l+1} for the $(l + 1)^{th}$ layer

Suppose we know the error, δ^{l+1}, at the $(l + 1)^{th}$ layer, then, when we apply the transpose matrix, $(w^{l+1})^T$, we can think intuitively of this moving ***backward*** through the network, giving us some measure of the error at the output of the l^{th} layer.

We then take the Hadamard product $\odot \sigma'(z^l)$, which moves the ***error backward through the activation function*** in layer l, giving us the δ^l in the weighted input to layer l.

BP-3: Equation for the rate of change of the cost wrt to any bias in network

In Particular:

$$\delta C/\delta b^l_j = \delta^l_j \qquad \text{(BP-3)}$$

That is, the error, δ^l_j, is exactly equal to the rate of change, $\delta C/\delta b^l_j$. This is great news, since BP-1 & BP-2 have already shown us how to compute δ^l_j. We can write BP-3 in shorthand as:

$$\delta C/\delta b = \delta$$

where it is understood that δ is being evaluated at the same neuron as the bias b.

BP-4: Equation for the rate of change of the cost wrt any weight in network

In particular:

$$\delta C/\delta w^l_{jk} = a^{l-1}_k \, \delta^l_j \qquad \text{(BP-4)}$$

This tells us how to compute the partial derivatives $\delta C/\delta w^l_{jk}$ in terms of the quantities a^{l-1}_k and δ^l_j, which we already know how to compute. The equation can be rewritten in a less index-heavy notation

$$\delta C/\delta w = a_{in} \, \delta_{out}$$

where it's understood that a_{in} is the activation of the neuron input to the weight w, and δ_{out} is the error of the neuron ouput from the weight w.

Slow Learning and Saturated Neurons

A nice consequence of the above equation is that when the activation a_{in} is small, i.e. $a_{in} \sim 0$, the gradient $\delta C/\delta w$ will also tend to be small. In this case, we'll say the weight ***learns slowly***, meaning that it's not changing much during gradient descent. In other words, one consequence of BP-4 is that weights output from low-activation neurons learn slowly.

Consider the term $\sigma'(z^L_j)$ in BP-1. Recall from the graph of the sigmoid function that the σ function becomes very flat when $\sigma(z^L_j)$ is approximately 0 or 1.

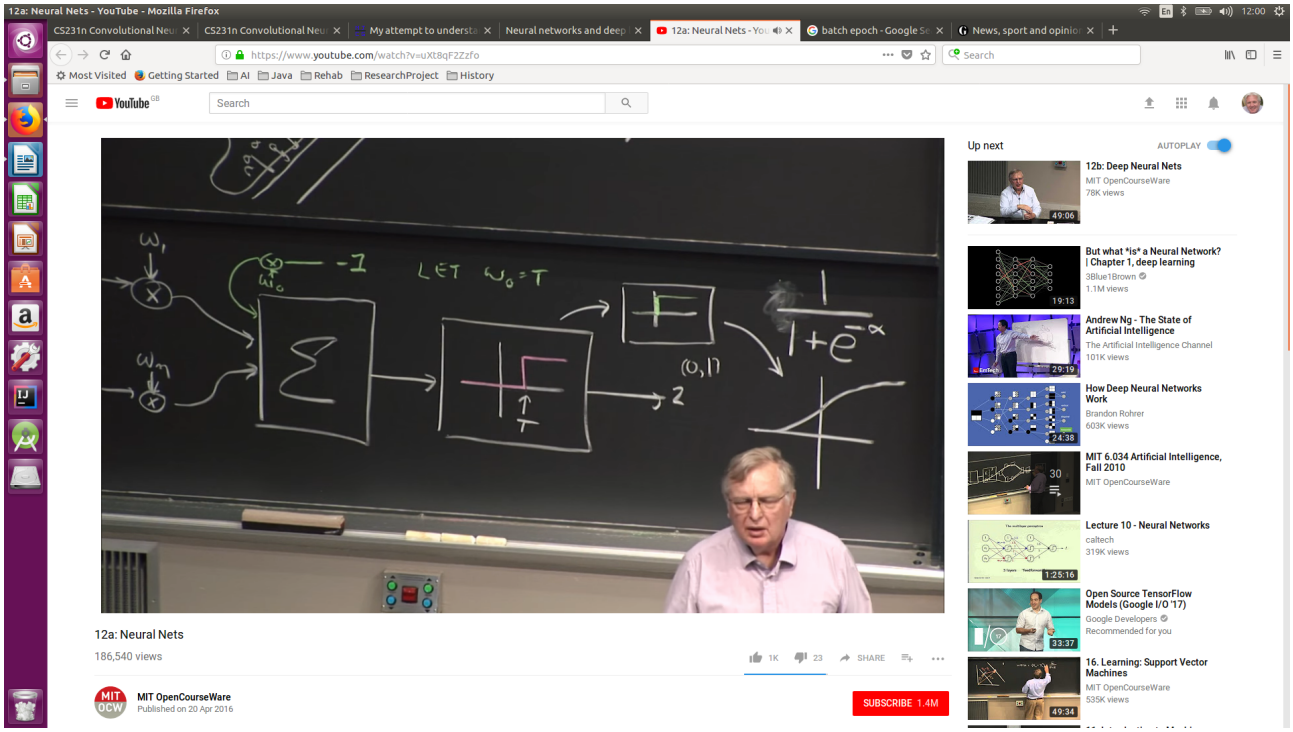


Illustration 18: Sigmoid Function

When this occurs we will have $\sigma'(z^L_j) \sim 0$ and so the lesson is that a weight in the final layer will learn slowly if the output neuron is either low activation (~0) or high activation (~1). In this case it is common to say the output neuron has ***saturated*** and, as a result, the weight has stopped learning (or learning slowly). Similar remarks hold also for biases of output neurons.

Summing up, we've learnt that a weight will learn slowly if either the ***input neuron*** is low-activation, or if the ***output*** has saturated, i.e. is either high- or low- activation.

The four fundamental equations turn out to hold for any activation function, not just the standard sigmoid function. And so we can use these equations to ***design*** activation functions, e.g. Rectified Linear Units (ReLU), which have particuar desired learning properties. As an example, suppose we were to choose an activation function σ so that σ' is always positive, and never gets near to zero. That would prevent the slow-down of learning that occurs when ordinary sigmoid functions saturate.

The backpropagation Algorithm

The back-propagation equations provide us with a way of computing the gradient of the cost function:

1. **Input x**:
   Set the corresponding activation a^l for the input layer
2. **Feedforward:**
   For each l = 2,3,…,L compute (or each Node in Graph)
   - $z^l = w^l a^{l-1} + b^l$
   - $a^l = \sigma(z^l)$
3. **Output error δ^L:**
   Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$ (BP-1)
4. **Backpropagate the error:**
   For each l = L – 1, L – 2, …, 2 (or Node in Graph)
   Compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$ (BP-2)
5. **Output:**
   The gradient of the cost function is given by:
   - $\delta C/\delta b^l_j = \delta^l_j$ (BP-3)
   - $\delta C/\delta w^l_{jk} = a^{l-1}_k \delta^l_j$ (BP-4)

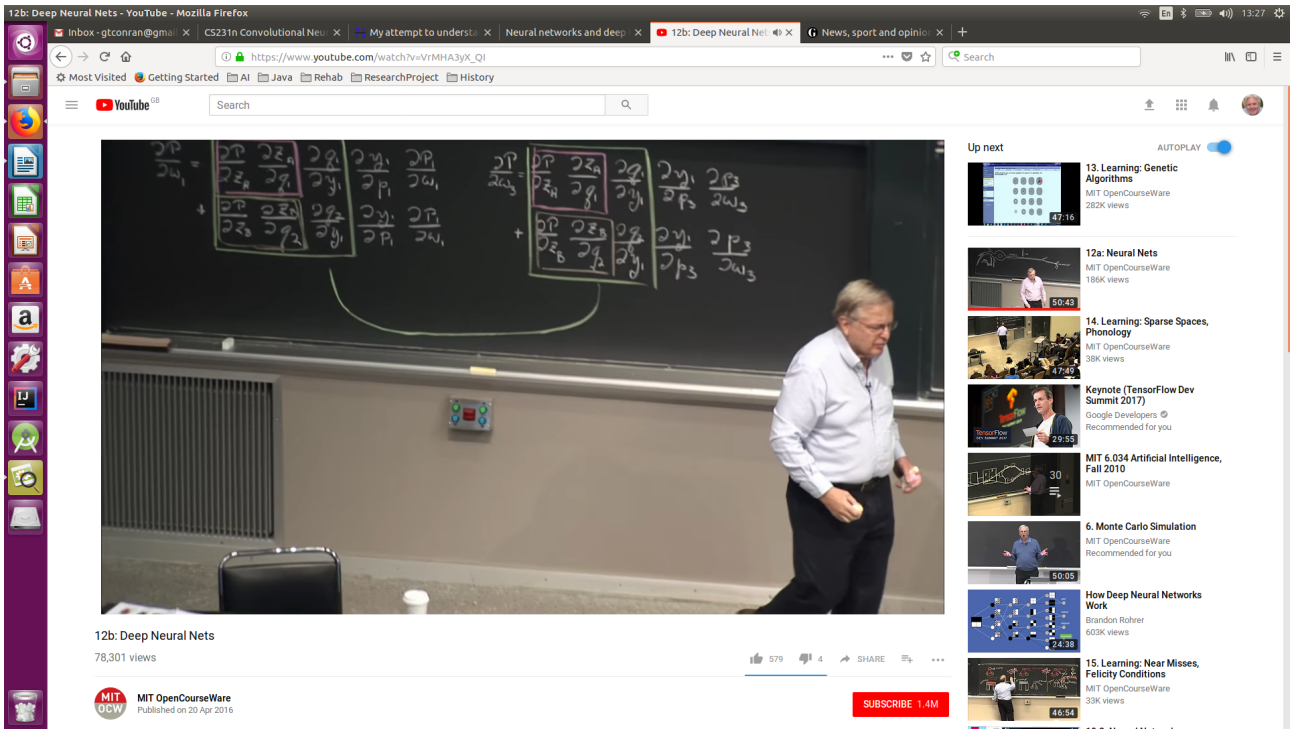


Illustration 19: Redundancy in Partial Derivatives makes Back-Propagataion Algorithm Computationally Tractable

2012: AlexNet: Ground Breaking use of CNNs for computer vision

ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, Ilya Sutskever, Geoff Hinton

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012 was the coming out party for CNNs in the computer vision community. This was the first time a model performed so well on the historically difficult ImageNet dataset. Utilizing techniques that are still used today, such as data augmentation and dropout, this paper really illustrated the benefits of CNNs and backed them up with record breaking performance in the competition.

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of ***five convolutional layers***, some of which are followed by ***max-pooling layers***, and ***three fully-connected layers*** with a ***final 1000-way softmax***. To make training faster, we used ***non-saturating neurons*** and a very efficient ***GPU implementation*** of the convolutional operation. To ***reduce overfitting*** in the fully-connected layers we employed a recently-developed regularisation method called "***dropout***" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Introduction

To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs) constitute one such class of models. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feed forward neural networks with similarly-sized models, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-bset performance is likely to be only slightly worse.

The specific contributions of this paper are:

- Trained one of the largest convolutional neural networks to date (2012) on the subsets of ImageNet used in ILSVRC-2010 and ILSVRC-2012 competitions and achieved by far the best results ever reported on these datasets.
- Wrote a highly-optimised GPU implementation of 2D convolution and all the other operations inherent in training convolutional neural networks (http://code.google.com/p/cuda-convnet/)
- Our network contains a number of new and unusual features which improve its performance and reduce its training time
- The size of our network made ***overfitting a significant problem***, even with 1.2 million labeled training examples, so we used several effective techniques for preventing overfitting
- Our final network contains five convolutional and three fully-connected layers, and this depth seems to be important: we found that removing any convolutional layer(each of which contains no more than 1% of the model's performance) resulted in inferior performance.

In the end, the network's size is limited mainly by the amount of memory available on current GPUs and by the amount of training time that we are willing to tolerate. All our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available.

The Dataset

ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories. The images were colletcted from the web and labeled by human labelers using Amazon mechanical Turk crowd-sourcing tool.

Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge, ILSVRC, has been held. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images.

On ImageNet, it is customary to report two error rates: top-1 and top-5.

ImageNet consists of variable-resolution images, which our system requires a constatnt input dimensionality. Therefore, we down-sampled the images to a fixed resolution of 256 x 256. We then trained our network on the (centered) raw RGB values of the pixels.

The Architecture

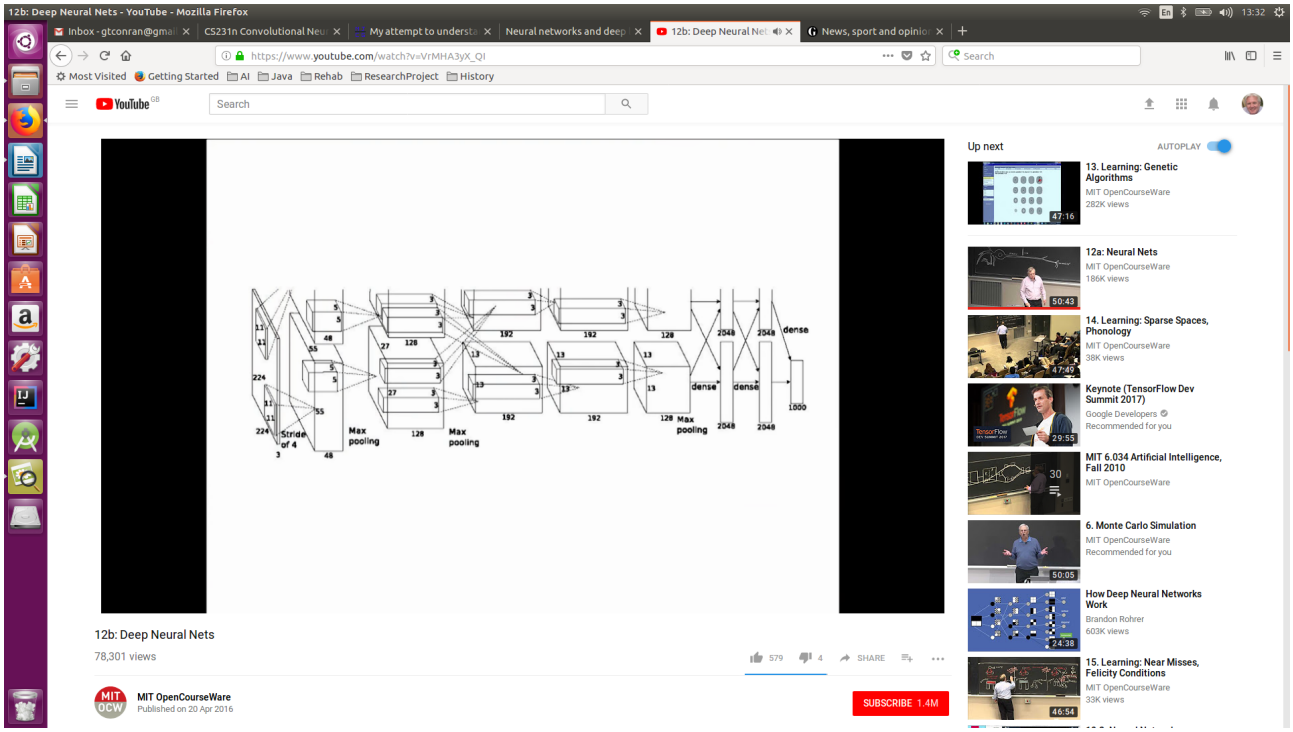


Illustration 20: AlexNet Architecture

AlexNet contains eight learned layers:

- 5 convolutional
- 3 fully connected

The output of the last fully-connected layer is fed to a 1000-way softmax which produces a distribution over the 1000 class labels. Our network maximises the multi-nomial logistic regression objective, which is equivalent to maximising the average across training cases of the log-probability of the correct label under the prediction distribution.

Below, we describe some of the novel or unusual features of the architecture:

ReLU Non-linearity

The standard way to model a neuron's output f as a function of its input x is with

$$f(x) = \tanh(x) \qquad \text{or} \qquad f(x) = (1 + e^{-x})^{-1}$$

In terms of training time with gradient descent, these ***saturating non-linearities*** are much slower than the ***non-saturating non-linearity***:

$$f(x) = \max(0, x)$$

We refer to neurons with this non-linearity as Rectified Linaer Units (ReLUs). Deep CNNs with ReLUs train several times faster than their equivalents with tanh units.

Training on Multiple GPUs

We spread the net across two GPUs. The parallellisation scheme used puts half of the kernels (or neurons) on each GPU but the GPUs communicate only in certain layers.

Local Response Normalisation

ReLUs have the desirable property that they do not require normalisation to prevent them form saturating. However, we still find that using a local normalisation scheme aids generalisation.

Overlapping Pooling

Pooling layers in CNNs summarise the outputs of neighbouring groups of neurons in the same kernel map. Traditionally, the neighbourhoods summarised by adjacent pooling units do not overlap. Overlapping pooling reduces the top-1 and top-5 error rates by 0.4% and 0.3% respectively, as compared to with non-overlapping schemes, which produces output of equivalent dimensions. We generally observe during training that models with overlapping pooling find it slightly more difficult to overfit.

Overall Architecture

The kernels of the second, fourth, and fifth convolutional layers are connected only to those kernel maps in the previous layer which reside on the same GPU. The kernels of the 3rd convolutional layer are connected to all kernel maps in the 2nd layer. The neurons in the FC layers are connected to all neurons in the previous layer. Response-normalisation layers follow the first and second convolutional layers. Max-pooling layers follow both response-normalisation layers as well as the fifth convolutional layer. The ReLU non-linearity is applied to the ouput of every convolutional and FC layer.

The 1st convolutional layer filters the 224 x 224 x 3 input image with 96 kernels of size 11 x 11 x 3 with a stride of 4 pixels. The 2nd convolutional layer takes as input the output of the 1st convolutional layer and filters it with 256 kernels of size 5 x 5 x 48. The 3rd, 4th, and 5th covolutional layers are connected to one another without any intervening pooling or normalisation layers. The 3rd convolutional layer has 384 layers of size 3x3x256 connected to the (normalised, pooled) outputs of the 2nd convolutional layer. The 4th convolutional layer has 384 kernels of size 3 x 3 x 192, and the 5th convolutional layer has 256 kernels of size 3 x 3 x 192. The fully-connected layers have 4096 neurons each.

Reducing Overfitting

Our neural network architecture has 60 million parameters. Below are two ways in which we combatted overfitting.

Data Augmentation

Easiest way to reduce overfitting on image data is to artifically enlarge the dataset using label-preserving transformations such as image translations, horizontal reflections, and patch extractions.

Dropout

Combining the predictions of many diferent models is a very succesful way to reduce test errors, but it appears to be too expensive for big neural networks that already take several days to train. There is, however, a very efficient version of model combination that only costs about a factor of two during training. The technique of “drop-out” consists of setting to zero the output of each hidden neuron with probability 0.5. The neurons dropped out in this way do not contribute to the forward pass and do not participate in back-propagation. So every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.

We use dropout in the first two FC layers. Without drop-out, our network exhibits substantial overfitting. Dropout roughly doubles the number of iterations required to converge.

Details of Learning

We trained our model using stochastic gradient descent with a batch size of 128 examples, momentum of 0.9, and weight decay of 0.0005. We found that this small amount of weight decay was important for the model to learn.

The update rule for weight w was:

$v_{i+1} := 0.9 \cdot v_i - 0.0005 \cdot \varepsilon \cdot w_i - \varepsilon \cdot \text{avg}(\partial L / \partial w_i)$

$w_{i+1} := w_i + v_{i+1}$

where:

- i is the iteration index
- v is the momentum variable
- ε is the learning rate
- $\text{avg}(\partial L / \partial w)$ is the average over the i-th batch of the derivative of the objective wrt w evaluated at w_i.

We initialised the weights in each layer from a zero-mean Gaussian distribution with SD 0.01. We initialised the neuron biases in the 2nd, 4th, and 5th convolutional layers, as well as the FC hidden layers, with the constant 1. We initilaied the neuron biases in the remaining layers with zero.

Results

Our network achieved top-1 and top-5 test error rates of 37.5% and 17.0%, far better than 2nd place which had 47.1% and 28.2% using a sparse-coding model.

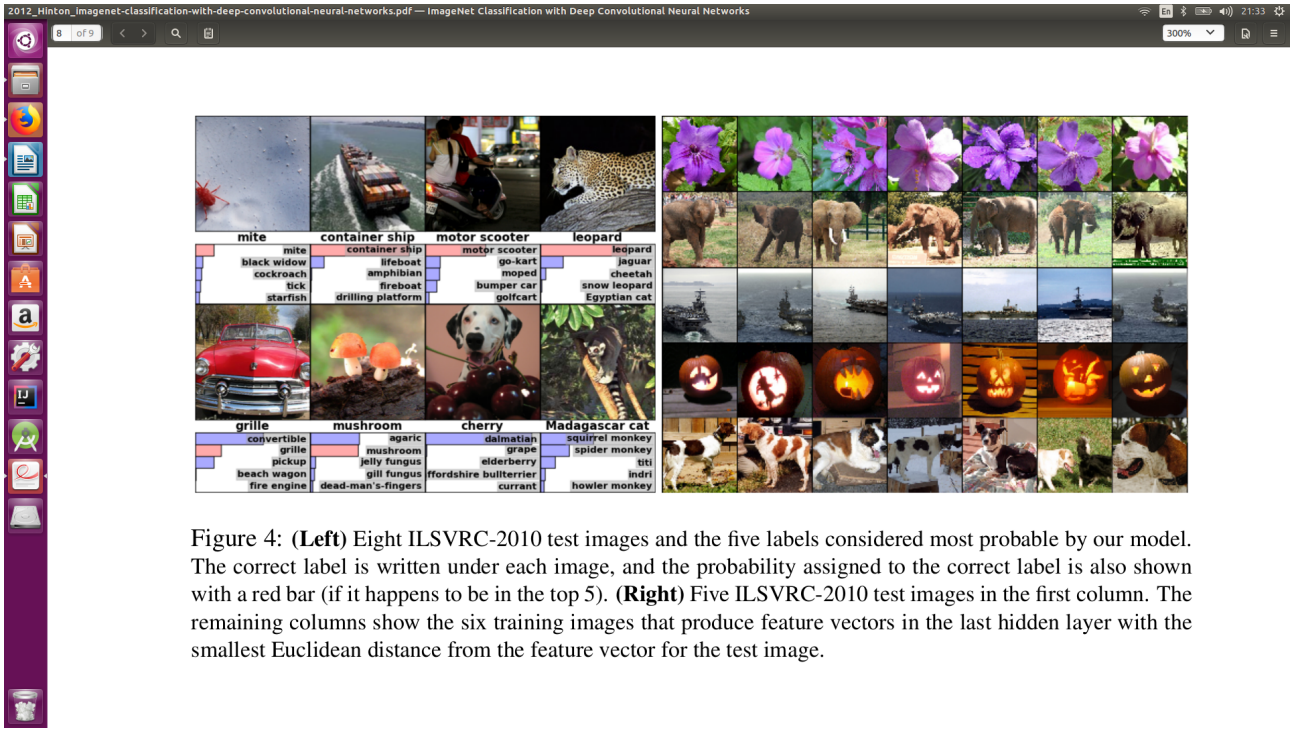


Figure 4: **(Left)** Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). **(Right)** Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Illustration 21: AlexNet Results

Discussion

A large deep, convolutional neural network is capable of achieving record breaking results on a highly challenging dataset using purely supervised learning. It is notable that the network's performance degrades if a single convolutional layer is removed. To simplify the experiments, we did not use any unsupervised pre-training, like clustering or PCA, even though it would probably help. The results show that improvements are to be had if we made our network bigger and trained it longer.

2014: ZF Net → Width Matters

Overfeat: Integrated Recognition, Localisation and Detection using Convolutional Networks

Abstract

We present an integrated framework for using Convolutional Networks for ***classification, localisation, and detection***. We show how a ***multi-scale and sliding window approach*** can be efficiently implemented within a ConvNet. We also introduce a novel deep learning approach to localisation by learning to predict object boundaries. ***Bounding boxes*** are then accumulated rather than suppressed in order to increase detection confidence. We show that different tasks can be learned simultaneously using a single shared network. This integrated framework is the winner of the localisation task of the ImageNet Large Scale Visual Recognition Challenge 2013 (ILSVCRC2013) and obtained very competitive results for the detection and classification tasks. In post-competition work, we establish a new state of the art for the detection task. Finally, we release a feature extractor from our best model called Overfeat.

2014: Inception V1 → Parameter Reduction

Going deeper with convolutions

Abstract

We propose a deep convolutional neural network architecture codenamed ***Inception***, which was responsible for setting the new state of the art classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilisation of the computing resources inside the network. This was achieved by carefully crafted design that allows for ***increasing the depth and width of the network*** while keeping the ***computational budget constant***. To optimise quality, the architectural decisions were based on the ***Hebbian principle*** and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called ***GoogLeNet***, a 22 layer deep network, the quality of which is assessed in the context of classification and detection.

Introduction

Our GoogLeNet submission to ILSVRC 2014 actually uses ***12x fewer parameters*** than the winning architecture Krizhevsky et al for 2012, while being significantly more accurate. The biggest gains on object-detection have not come from the utilisation of deep networks alone or bigger models, but from the ***synergy of deep architectures and classical computer vision***, like Girshick's R-CNN algorithm.

Another notable factor is that with the ***ongoing traction of mobile and embedded computing***, the efficiency of our algorithms – especially their power and memory use – gains importance.

The focus of the paper is on an efficient deep neural network architecture for computer vision, codenamed Inception. The work "deep" is used in two different meanings:

- we introduce a new level of organisation in the form of the "Inception module"
- the more direct sense of increased network depth

Related Work

Starting with LeNet-5, convolutional neural networks (CNN) have typically had a standard structure – stacked convolutional layers are followed by one or more FC layers. For larger datasets, such as ImageNet, the trend has been to increase the number of layers and layer size, while supporting drop-out to addres the problem of overfitting.

Serre et al.use a series of ***Gabor filters*** of different sizes to handle multiple scales, similarly to the Inception model. However, contrary to the fixed 2-layer deep model of Serre et al., ***all filters in the Inception model are learned***. Further more, Inception layers are repeated many times, leading to a 22-layer deep model in the case of the GoogleNet model.

Network-in-Network is an approach that increases the representational power of neural networks. When applied to convolutional layers, the method could be viewed as additional ***1x1 convolutional layers*** followed typically by the rectified linear activation. This enables it to be easily integrated in current CNN pipelines. This approach is used heavily within the Inception architecture. Critically, they are used as ***dimension reduction modules*** to remove computational bottlenecks, that would otherwise limit the size of networks. This allows for not just ***increasing the depth, but also the width of our networks without significant performance penalty***.

Motivation & High Level Considerations

The easiest way to improve the performance of deep neural networks is by increasing their size, i.e. network width and depth. However, this simple approach has two major drawbacks:

- Bigger size typically means a larger number of parameters, leading to over-fitting
- Dramatic increased use of computational resources

The fundamental way of solving both issues is to move from fully connected to sparsely connected architectures, even ***inside convolutions***. Arora et al. noted that if the probability distribution of the data-set is representable by a large, very ***sparse deep neural network***, then the ***optimal network topology*** can be constructed layer by layer by analyzing the correlation statistics of the activations of the last layer and ***clustering neurons with highly correlated outputs***. This resonates with the well known ***Hebbian principle*** – neurons that fire together, wire together.

On the downside, today's computing infrastructures are very inefficient when it comes to numerical calculation on ***non-uniform sparse data structures***. Even if the number of arthimetic operations is reduced by 100x, the overhead lookups and cache misses is so dominant that switching to sparse matrices would not pay off.

The Inception architecture tries to address these two issues. It is an algorithm that tries to approximate a ***sparse structure*** for vision networks but covers the hypothetical outcome by using ***dense, readily available components***.

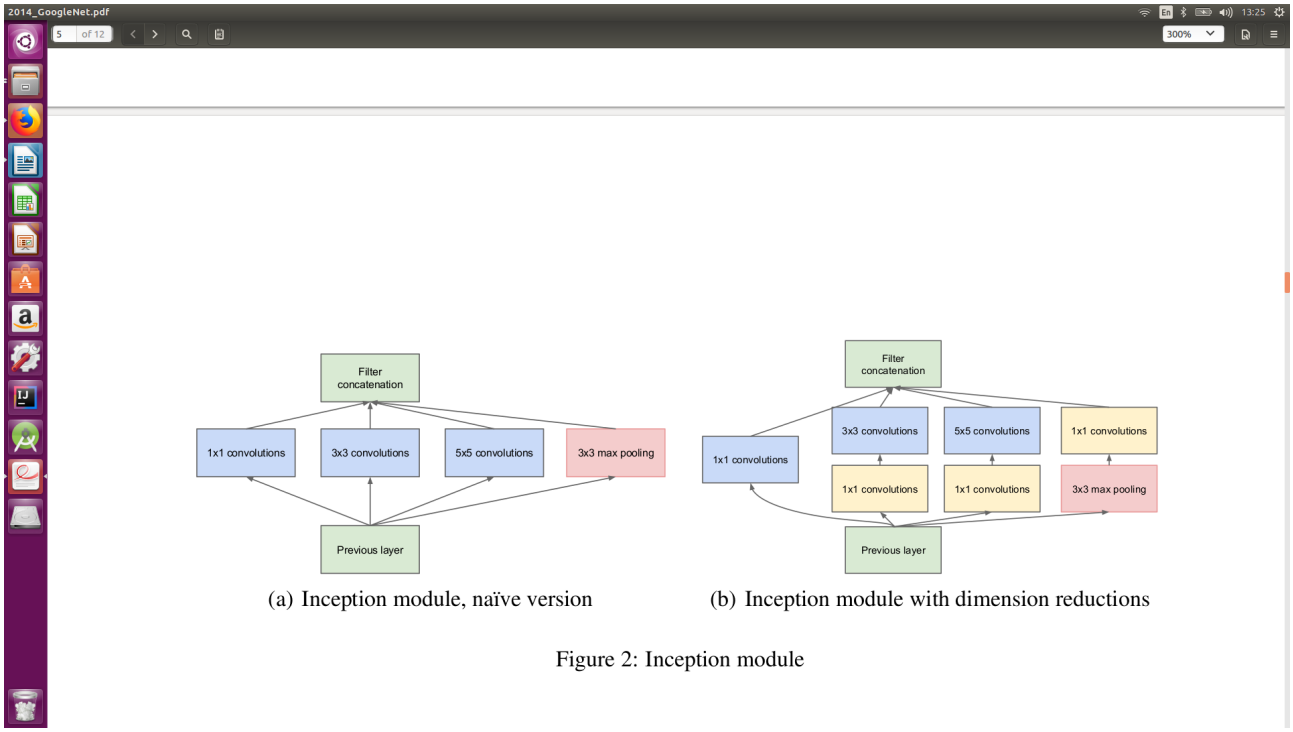
Architectral Details

As mentioned above, the ***main idea*** of the Inception architecture is based on finding out how an optimal local ***sparse structure*** in a convolutional vision network can be approximated and covered by readily available ***dense components***. Note that assuming translation invariance means our network will be built from convolutional building blocks. ***All we need to do is to find the optimal local construction and to repeat it spatially***.

As Inception modules are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease suggesting that the ratio of 3x3 and 5x5 convolutions should increase as we move to higher levels.

Unfortunately, even a modest number of 5x5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters.

This leads to the ***second idea*** behind the Inception architecture: judiciously applying ***dimension reductions and projections*** wherever the computational requirements would increase too much otherwise.



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

Figure 2: Inception module

Illustration 22: Inception Module

A main benefit of this architecture is that it allows for increasing the number of units at each stage significantly without an uncontrolled blow-up in computational complexity. An additional benefit is that it aligns with the intuition that visual information should be processed at various scales (think frequencies in time-freq analysis) and then aggregated so that the next stage can abstract features from different scales similtaneously.

2015: Inception V2 → Batch Normalisation

Batch Normalisation: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialisation, and makes it notoriously hard to train models with saturating non-linearities. We refer to this phenomeon as ***internal covariate shift***, and address the problem by normalising layer inputs. Our method draws its strength from ***making normalisation a part of the model architecture*** and performing the normalisation ***for each training mini-batch***. Batch Normalisation allows us to use much higher learning rates and be less careful about initialisation. It also acts as a regulariser, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalisation achieves the same accuracy with 14 times fewer training steps, and beats the original model by a signficant margin. Using an ***ensemble of batch-normalised networks***, we improve upon the best publised result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

Summary

(https://medium.com/deeper-learning/glossary-of-deep-learning-batch-normalisation-8266dcd2fa82)

Batch normalisation is a technique for improving the performance and stability of neural networks, and also makes more sophisticated deep learning architectures work in practice (like DCGANs).

The idea is to normalise the ***inputs of each layer*** in such a way that they have a mean output activation of zero and standard deviation of one. This is analogous to how the inputs to networks are standardised.

Benefits of Batch Normalization

The intention behind batch normalisation is to optimise network training. It has been shown to have several benefits:

1. **Networks train faster**—Whilst each training iteration will be slower because of the extra normalisation calculations during the forward pass and the additional hyperparameters to train during back propagation. However, it should converge much more quickly, so training should be faster overall.
2. **Allows higher learning rates**—Gradient descent usually requires small learning rates for the network to converge. As networks get deeper, gradients get smaller during back propagation, and so require even more iterations. Using batch normalisation allows much higher learning rates, increasing the speed at which networks train.

3. **Makes weights easier to initialise**—Weight initialisation can be difficult, especially when creating deeper networks. Batch normalisation helps reduce the sensitivity to the initial starting weights.
4. **Makes more activation functions viable**—Some activation functions don't work well in certain situations. Sigmoids lose their gradient quickly, which means they can't be used in deep networks, and ReLUs often die out during training (stop learning completely), so we must be careful about the range of values fed into them. But as batch normalisation regulates the values going into each activation function, nonlinearities that don't work well in deep networks tend to become viable again.
5. **Simplifies the creation of deeper networks**—The previous 4 points make it easier to build and faster to train deeper neural networks, and deeper networks generally produce better results.
6. **Provides some regularisation**—Batch normalisation adds a little noise to your network, and in some cases, (e.g. Inception modules) it has been shown to work as well as dropout. You can consider batch normalisation as a bit of extra regularization, allowing you to reduce some of the dropout you might add to a network.

As batch normalisation helps train networks faster, it also facilitates greater experimentation—as you can iterate over more designs more quickly.

2015: Inception V3 → tweaked for Mobile & Big Data

Rethinking the Inception Architecture for Computer Vision

Abstract

Convolutional networks are at the core of most state-of-the-art computer vision solutions for a wide variety of tasks. Since 2014 very deep convolutional networks started to beocme mainstream, yielding substantial gains in various benchmarks. Although increased model size and computational cost tend to translate to immediate quality gains for most tasks (as long as enough labeled data is provided for training), computational efficiency and low parameter count are still enabling factors for various use cases such as ***mobile vision and big-data*** scenarios. Here we are exploring ways to scale up networks in ways that aim at ***utilising the added computation as efficiently as possible*** by suitably ***factorised convolutions and aggressive regularisation.*** We benchmark our methods on the ILSVRC20102 classification challenge validation set demonstrate substantial gains over the state of the art: 21.2% top-1 and 5.6% top-5 error for single frame evaluation using a network with a computational cost of 5 billion multiply-adds per inference and with using less than 25 million parameters. With an ensemble of 4 models and multi-crop evaluation, we report 3.5% top-5 error and 17.3% top-1 error.

Introduction

In this paper, we start with describing a few general principles and optimisation ideas that proved to be useful for scaling up convolution networks in efficient ways. Although our principles are not limited to Inception-type networks, they are easier to observe in that context as the generic structure of the Inception style building blocks is flexible enough to incorporate those constraints naturally. This is enabled by the generous use of ***dimensional reduction*** and ***parallel structures*** of the Inception modules which allows for mitigating the impact of structural changes on nearby components.

General Design Principles

- Avoid representational bottlenecks
- Higher dimensional representations are easier to process locally within a network
- Spatial aggregation can be done over lower dimensional embeddings without much or any loss in representational power
- Balance the width and height of the network

Factorising Convolutions with Large Filter Size

Much of the original gains of the GoogLeNet network arise from a very generous use of dimension reduction. This can be viewed as a special case of factorising convolutions in a computationally

efficient manner. Consider for example the case of a 1x1 convolutional layer followed by a 3x3 convolutional layer. In a vision network, it is expected that the outputs of near-by activations are highly correlated. Therefore, we can expect that their activations can be reduced before aggregation and that this should result in similarly expressive local representations. Here are other ways of factorising convolutions in various settings:

- Factorisation into smaller convolutions
- Spatial Factorisation into Asymmetric Convolutions

Utility of Auxiliary Classifiers

The original motivation of auxiliary classifiers was to push useful gradients to the lower layers to make them immediately useful and improve the convergence during training by combating the vanishing gradient problem in very deep networks. They also promote more stable learning and better convergence. Auxiliary classifiers act as ***regularisers*** because the main classifier of the network performs better if the side branch is ***batch-normalised*** or has a drop-out layer.

Efficient Grid Size Reduction

Inception-v2

2016: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

Abstract

Very deep convolutional networks have been central to the largest advances in image recognition performance in recent years. One example is the Inception architecture that has been shown to achieve very good performance at relatively low computational cost. Recently, the introduction of residual connections in conjunction with a more traditional architecture has yielded state-of-the-art performance in the 2015 ILSVRC challenge; its performance was similar to the latest generation Inception-v3 network. This raises the question of whether there are any benrfits in combining the Inception architecture with residual connections. Here we give clear empirical evidence that training with residual connections accelerates the training of Inception networks significantly. There is also some evidence of residual Inception networks outperforming similarly expensive inception networks without residual connections by a thin margin. We also present several new streamlined architectures for both residual and non-residual Inception networks. These variations improve the single-frame recognition performance on the ILSVRC 2012 classification task significantly. We further demonstrate how proper activation scaling stabilises the training of very wide residual inception networks. With an ensemble of three residual and one Inception-v4, we achieve 3.08% top-5 error on the test set of the ImageNet classification (CLS#0 challenge.

2015: VGGNet / OxfordNet → Depth Matters

Very Deep Convolutional Networks for Large-Scale Image Recognition

Abstract

In this work we investigate the effect of the convolutional network depth on its accuracy in the large-scale recognition setting. Our main contribution is a thorough avaluation of networks of increasing depth using an architecture with very small (3x3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers. These findings were the basis of our ImageNet Challenge 2014 submission, where aur team secured the first and second places in the localisation and classification tracks respectively. We also show that our representations generalise well to other datasets, where they achieve state-of-the-art results. We have made our two best-performing ConvNet models publicly available to facilitate further research on the use of deep visual representations in computer vision.

2013 - Transfer Learning

DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition

Abstract

We evaluate whether features extracted from the activation of a deep convolutional ***network trained in a fully supervised fashion on a large, fixed set of object recognition tasks can be re-purposed to novel generic tasks***. Our generic tasks may differ significantly from the originally trained tasks and there may be insufficient labeled or unlabeled data to conventionally train or adapt a deep architecture to the new tasks. We investigate and visualise the semantic clustering of deep convolutional features wrt a variety of tasks, including ***scene recognition***, ***domain adaptation***, and ***fine-grained recognition*** challenges. We compare the efficacy of relying on various network levels to define a fixed feature, and report novel results that significantly outperform the state-of-the-art on several important vision challenges.

Introduction

Deep models have been applied to large-scale visual recognition tasks, trained via back-propagation through layers of convolutional filters, LeCun et al, 1998 and by 2012, with Krizhevsky et al, they have outperformed all known methods on a large scale challenge.

With limited training data, however, fully-supervised deep architectures will generally dramatically overfit the training data.

This paper investigates semi-supervised multi-task learning of deep convolutional representations, where representations are learned on a set of related problems but applied to new tasks which have too few training examples to learn a full deep representation. The model can either be considered as:

- a deep architecture for ***transfer learning*** based on a supervised pre-training phase
- simply as a ***new visual feature DeCAF*** defined by the convolutional network weights learned on a set of pre-defined object recognition tasks

The main result is the empirical validation that a ***generic visual feature*** based on a convolutional network weights trained on ImageNet outperforms a host of conventional visual representations on standard benchmark object recognition tasks, including Caltech-101 and others.

We also found, by visualising semantic clustering properties, that convolutional features appear to cluster semantic topics more readily than conventional features.

Related Work

We investiagte the supervised pre-training approach using a concrpt-bank paradigm by learning the features on large-scale data in a supervised setting, then transfering them to different tasks with different labels.

Deep Convolutional Activation Features

A deep convolutional model is first trained in a fully supervised setting using AlexNet and we extract various features and evaluate the efficacy of these features on generic vision tasks.

We ask two questions:

- Do features extracted from the CNN generlaise to other datasets?
- How do these features perform versus depth?

We address these questions both qualitatively and quantatively, via visualisations of semantic clusters and experimental comparison to current baselines.

Feature Generalisation & Visualisation

We visualise features in the following way: we run ***t-SNE algorithm*** (Hinton, 2008) to find a 2-d embedding of the high-dim feature space, and plot them as points coloured depending on their semantic category in a particular hierarchy.

Experiments

- Object Recognition
- Domain adaptation
- Subcategory recognition
- Scenr recognition

Discussion

We demonstrate that by leveraging an auxiliary large labeled object database to train a deep convolutional architecture, we can learn features that have sufficient representational power and generlisation ability to perform semantic visual discrimination tasks using simple linear classifiers, reliably out-performing approaches based on sophisticated multi-kernel learning techniques with traditional hand-engineered features.

Our visual results demonstrate the generality and semantic knowledge implicit in these features, showing that the features tend to cluster images into interesting semantic categories on which the network was never explicitly trained.

Our numerical results consistently and robustly demonstrate that our multi-task feature learning framework can substantially improve the performance of a wide variety of existing methods across a spectrum of visual recognition tasks, including domain adaptation, fine-grained part-based recognition, and large-scale recognition.