

Uczenie głębokie (Deep Learning)

Module 0: Introduction

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2019–2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Course logistics

- 30 hrs of lectures, 30 hrs of labs
- Lectures: prof. dr hab inż. Krzysztof Krawiec
 - Grouped: 2 lectures, odd/even weeks
 - Occasionally swapped with other courses (will notify)
- Labs: mgr inż. Antoni Nowinowski
 - In general, not synchronized with the lectures
- Lecture slides gradually made available in eKursy
- Final exam:
 - Written
 - Scope: almost identical with the lectures (with some exclusions)
 - Several open/descriptive questions + a number of choice questions

Course objectives

Excerpt from the ECTS card:

1. Provide students with basic knowledge of selected topics of deep learning, neural networks, representation learning and related machine learning aspects.
2. Developing students' capability to solve problems and machine learning tasks with the use of deep learning models, including deep neural networks, especially classification, regression, representation learning and feature engineering.
3. Acquiring experience in the use of these skills in selected practical applications.
4. Development of skills of effective work on software implementation projects in the field of deep learning, including cooperation in small project groups.

Emphasis on understanding of the underlying concepts and mechanisms.

Bibliography (English and Polish)

In English:

1. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep learning*, MIT Press, 2016.
<https://www.deeplearningbook.org/>
2. Josh Patterson, Adam Gibson, *Deep learning: A Practitioner's Approach*. O'Reilly, 2017.
3. Francois Chollet, *Deep Learning with Python*, Manning Publications, 2018.

Polish translations:

1. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep learning: systemy uczące się*. Wydawnictwo Naukowe PWN, 2018.
2. Josh Patterson, Adam Gibson, *Deep learning: praktyczne wprowadzenie*. Grupa Wydawnicza Helion. 2018.
3. Francois Chollet, *Deep Learning: Praca z językiem Python i biblioteką Keras*, Helion 2019.

Deep learning book

Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep learning*, MIT Press, 2016.

DLB, <https://www.deeplearningbook.org/>

The Deep Learning textbook is a resource intended to help students and practitioners enter the field of machine learning in general and deep learning in particular. The online version of the book is now complete and will remain available online for free.

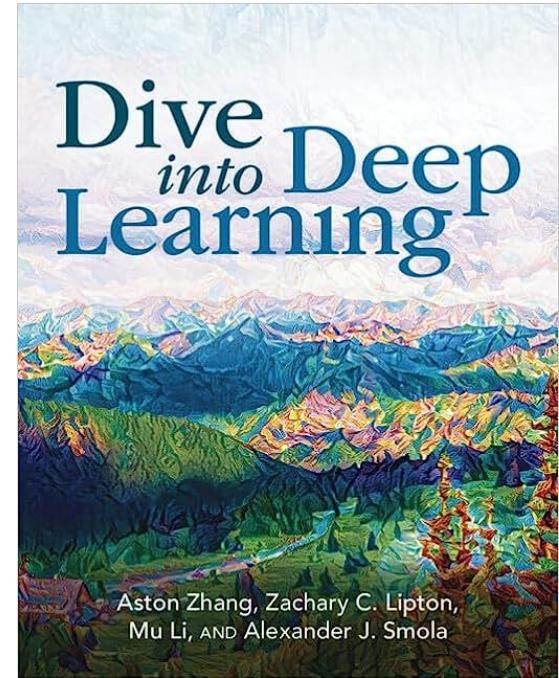
Still right about the principles, though a bit outdated in terms of classes of models presented.

An even better Deep Learning textbook?

- Aston Zhang, Zachary C. Lipton, Mu Li,
Alexander J. Smola,
*Dive into Deep Learning: Interactive deep
learning book with code, math, and discussions*
- Implemented with PyTorch, NumPy/MXNet, JAX,
and TensorFlow
- Adopted at 500 universities from 70 countries

<https://d2l.ai/index.html>

<https://github.com/d2l-ai/d2l-en>



Extended bibliography (in Polish)

1. Valentino Zocca, Gianmario Spacagna, Deep learning: uczenie głębokie z językiem Python: sztuczna inteligencja i sieci neuronowe, Grupa Wydawnicza Helion, 2018.
2. Krzysztof Krawiec Jerzy Stefanowski. Uczenie maszynowe i sieci neuronowe. Politechnika Poznańska. Wydawnictwo, 2003, 2004
3. Ryszard Tadeusiewicz. Sieci neuronowe. Akademicka Oficyna Wydawnicza RM, 1993.
4. Andrew W. Trask, Zrozumieć głębokie Uczenie, PWN 2019.
5. Valentino Zocca, Gianmario Spacagna, Daniel Slater, Peter Roelants, Deep Learning. Uczenie głębokie z językiem Python. Helion 2018.

Related:

1. Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong, Matematyka w uczeniu maszynowym, Helion 2022.
<https://helion.pl/ksiazki/matematyka-w-uczeniu-maszynowym-marc-peter-deisenroth-a-aldo-faisal-cheng-soon.mawuma.htm#format/d>

Modules (all)

- 0. Introduction
- 0.1. Introduction to neural networks.
- 0.2. What is deep learning?
- ~~1. Convolutional Neural Networks (basic architectures)~~
- 1.1. Convolutional Neural Networks: Advanced models
- 2. Generative Adversarial Networks
- 3. Autoencoders
- 4. Recurrent Neural Networks
- 5. Deep clustering
- 6. Graph Neural Networks
- 7. Point cloud models
- 8. Transformers
- 9. Theory and analysis
- 10. Selected topics

Case studies and applications

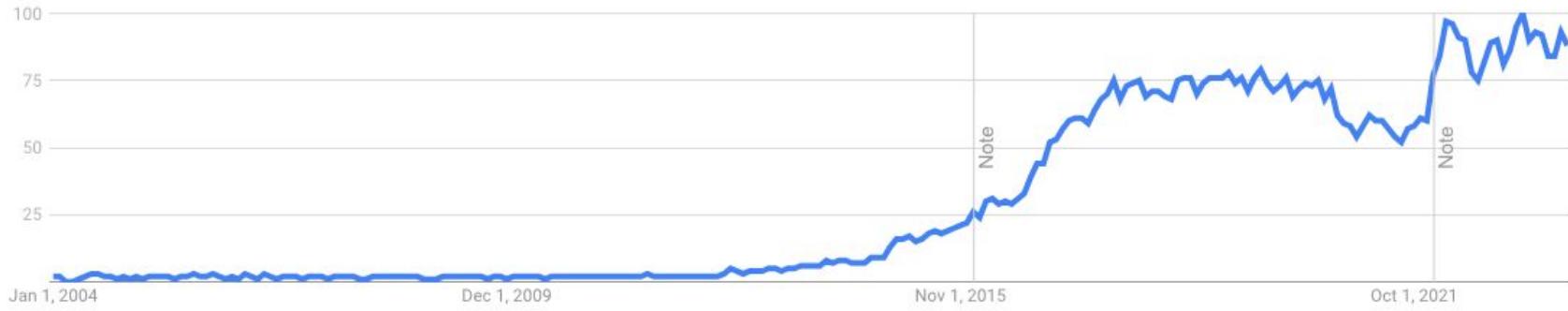
1. Image classification, object recognition, semantic segmentation
2. Unsupervised segmentation
3. Image denoising
4. Autoencoders for representation learning
5. Graph NNs for proof search and molecular discovery
6. Disentanglement of features
7. Program synthesis for scene understanding
8. Neuroevolution
9. ...

Remarks

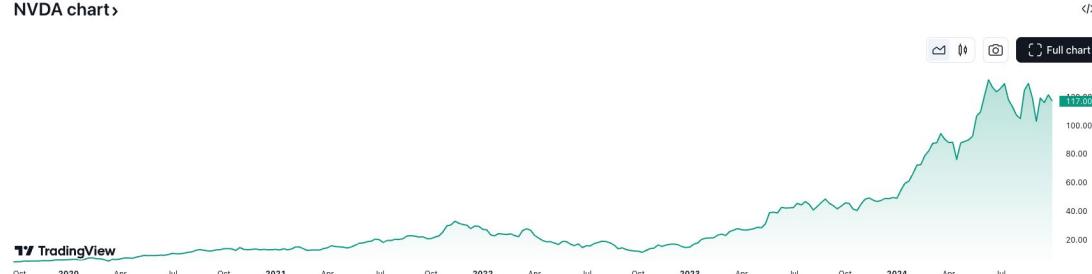
- Related topics and subjects:
 - machine learning, image processing, pattern recognition, computer vision, continuous optimization,
- Focus on the milestones in deep learning development
- The challenges:
 - the volume of available material
 - the rate of development
- Topics intentionally dropped from the course:
 - Reinforcement learning (RL)
 - Natural language processing (NLP)
 - Large Language Models (LLMs)
- More advanced topics will be not required at the exam
 - The detailed list will be provided in due course.

Popularity and demand

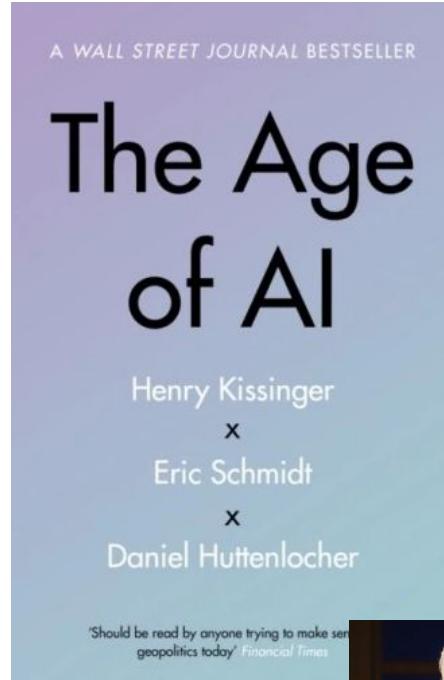
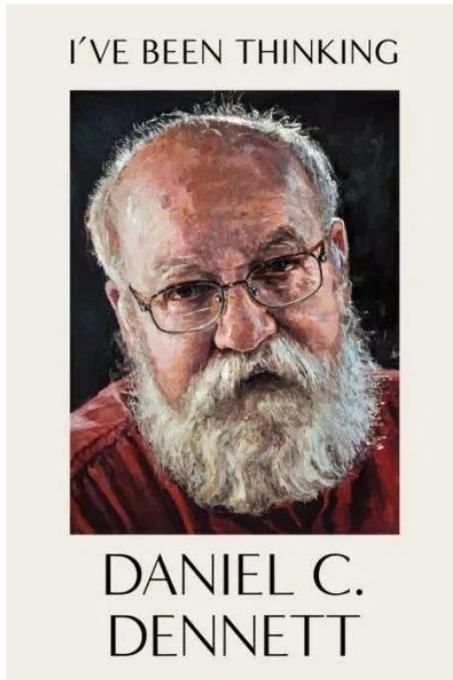
Popularity of the “deep learning” phrase according to Google Trends



NVDA chart >



Everybody is an expert in AI ...



End of module

Deep Learning

Module 0.1: Introduction to Neural Networks

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2019–2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Module outline

1. What is a neural network?
2. Units and layers
3. Computational capabilities of neural networks
4. Training neural networks
5. Gradient descent

What is a neural network?

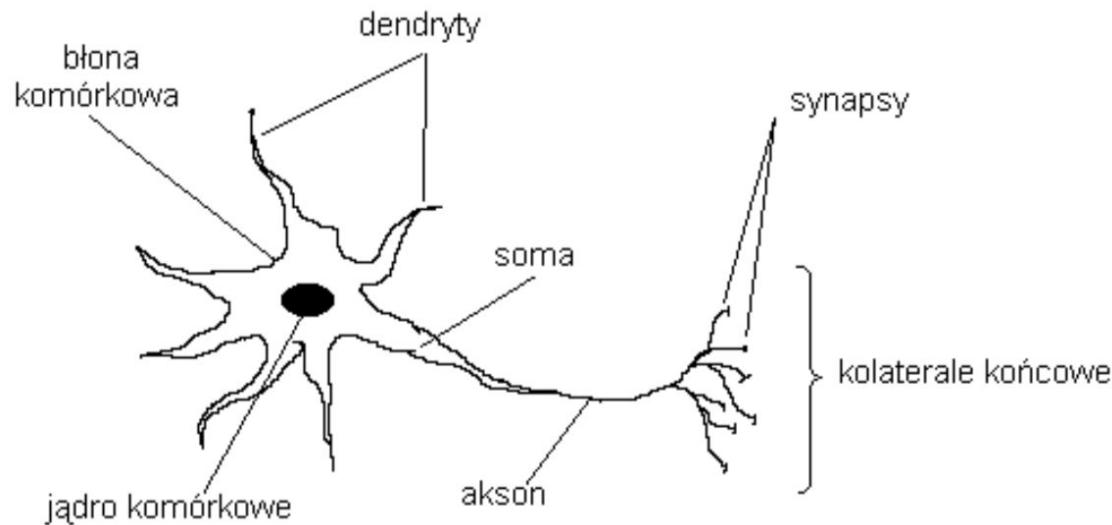
What is [an artificial] neural network?

Definition: A number of interconnected, relatively simple elementary processing units ('neurons').

- The expressive power stems from the architecture and the number of entities, not from their individual capabilities.
 - The 'natural' way of making a NN model more powerful is to equip it with more units.
- An alternative model of computation (bio-inspired, but very loosely).
- A machine learning model, which can be trained and queried.
- Model querying:
 - Input data fed into network (input units).
 - The data ('signals') propagate through the architecture
 - Collecting model's response from the output units.

Biological inspirations for ANNs

- More essential in the early days of the field, now largely forgotten.
 - (or found largely irrelevant).
- ANNs' units are very very crude models of biological neurons.



A bit of history

- Beginnings: 1940s (McCulloch and Pitts, 1943)
- **First spring:** Early days of AI
 - Perceptron: (Rosenblatt, 1958)
 - Hopfield networks
- **Second spring:** late 1980s – mid 1990s
 - Parallel Distributed Processing: Explorations in the Microstructure of Cognition, by Rumelhart, McClelland, and PDP Research Group, 1986,
- Then: Neural network winter
 - Saturation of capabilities combined with insufficient performance on many real-world tasks.
 - Growing popularity and performance of other ML methods (support vector machines (SVM), random forests, Bayesian models, etc.)
- **Third spring:** Big come-back: ~2005 and on
 - New wave: Deep Learning
 - Facilitated by conceptual developments, growing affordability of computing power and increasing availability of data.

Key figures

- Geoffrey Hinton
- Yann LeCun
- Juergen Schmidhuber
- Yoshua Bengio
- Michael Jordan
- Andrew Ng
- Ian Goodfellow
- Daphne Koller
- Andrej Karpathy
- Fei-Fei Li
- Sebastian Thrun
- Sepp Hochreiter
- ...

Fathers of the Deep Learning Revolution Receive ACM A.M. Turing Award

Bengio, Hinton and LeCun Ushered in Major Breakthroughs in Artificial Intelligence



Widespread recognition

The 2024 physics laureates

The Nobel Prize in Physics 2024 was awarded to John J. Hopfield and Geoffrey E. Hinton “for foundational discoveries and inventions that enable machine learning with artificial neural networks.”

John Hopfield created an associative memory that can store and reconstruct images and other types of patterns in data. Geoffrey Hinton invented a method that can autonomously find properties in data, and so perform tasks such as identifying specific elements in pictures.



John Hopfield and Geoffrey Hinton. Ill. Niklas Elmehed © Nobel Prize Outreach

The 2024 chemistry laureates

The Nobel Prize in Chemistry 2024 was awarded with one half to David Baker “for computational protein design” and the other half jointly to Demis Hassabis and John M. Jumper “for protein structure prediction”.

Demis Hassabis and John Jumper have successfully utilised artificial intelligence to predict the structure of almost all known proteins. David Baker has learned how to master life’s building blocks and create entirely new proteins.



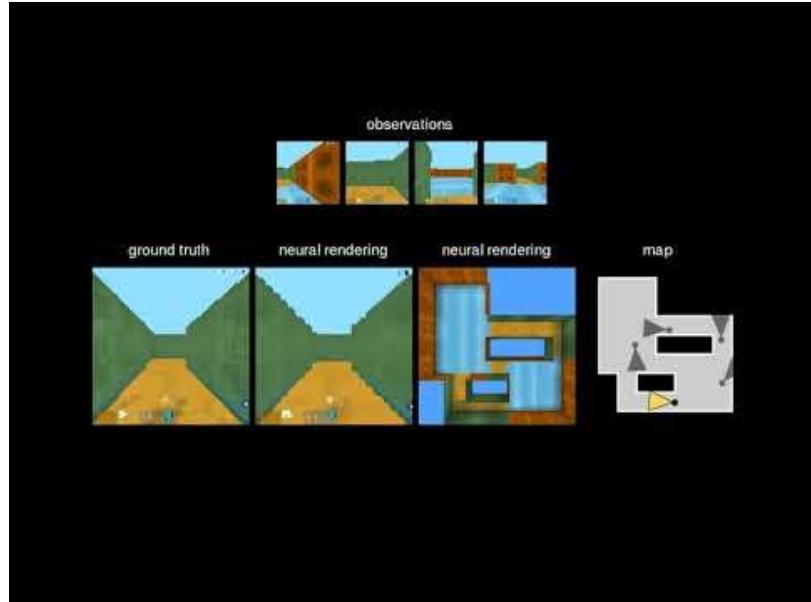
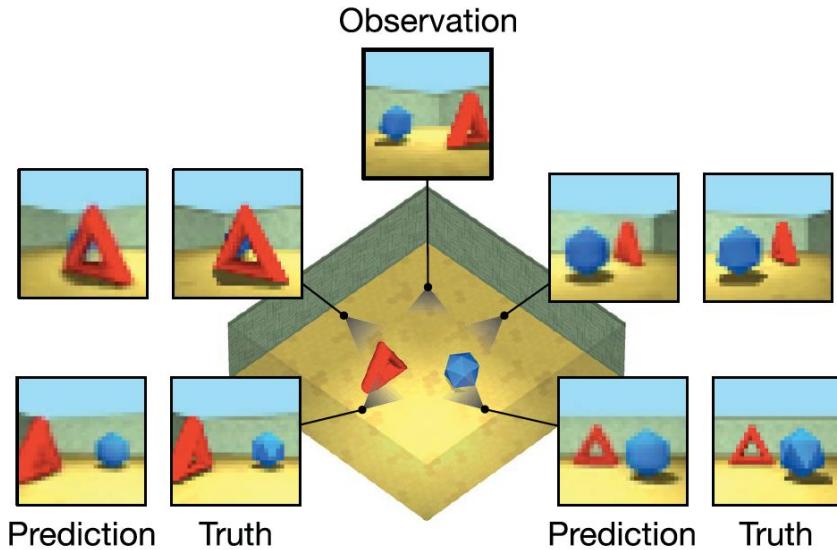
David Baker, Demis Hassabis and John Jumper. Ill. Niklas Elmehed © Nobel Prize Outreach

Current state of the domain

- Currently: one of the most successful paradigms of AI and machine learning (ML), particularly in:
 - image analysis, pattern recognition, computer vision,
 - natural language processing,
 - generative models.
- Brought the capability to handle (simulate, train, query) large, deep (many-layers) networks.
- Rich branch of research on the verge of AI, ML, and other disciplines (e.g., cognitive sciences)
- A broad range of approaches and architectures: probabilistic, evolutionary, spiking, counterpropagation, discrete, bidirectional, ...
- Handles surprisingly well different data representations, both:
 - fixed-size: vectors, matrices, images, tensors,
 - variable-size: sequences, trees, graphs, text, ...

A sample of capabilities

The model predicts the unknown view of an (unknown) scene based on a few known views.



A sample of capabilities

Translation between programming languages

Prompt

```
##### Translate this function from Python into Haskell
### Python

def predict_proba(X: Iterable[str]):
    return np.array([predict_one_probas(tweet) for tweet in X])

### Haskell
```

Sample response

```
predict_proba :: [String] -> [Probability]
predict_proba = map predict_one_probas
```

A sample of capabilities

Explaining a piece of code
in natural language.

Prompt

```
class Log:  
    def __init__(self, path):  
        dirname = os.path.dirname(path)  
        os.makedirs(dirname, exist_ok=True)  
        f = open(path, "a+")  
  
        # Check that the file is newline-terminated  
        size = os.path.getsize(path)  
        if size > 0:  
            f.seek(size - 1)  
            end = f.read(1)  
            if end != "\n":  
                f.write("\n")  
        self.f = f  
        self.path = path  
  
    def log(self, event):  
        event["_event_id"] = str(uuid.uuid4())  
        json.dump(event, self.f)  
        self.f.write("\n")  
  
    def state(self):  
        state = {"complete": set(), "last": None}  
        for line in open(self.path):  
            event = json.loads(line)  
            if event["type"] == "submit" and event["success"]:  
                state["complete"].add(event["id"])  
                state["last"] = event  
        return state
```

Sample response

The constructor creates a directory for the log file if it doesn't exist.
2. The log() method writes a JSON-encoded event to the log file.
3. The state() method returns a dictionary with the set of complete tasks and the most recent event.

A sample of capabilities

Generate Python docstrings and comments for a piece of code

Prompt

```
# Python 3.7

def randomly_split_dataset(folder, filename, split_ratio=[0.8, 0.2]):
    df = pd.read_json(folder + filename, lines=True)
    train_name, test_name = "train.jsonl", "test.jsonl"
    df_train, df_test = train_test_split(df, test_size=split_ratio[1],
                                         random_state=42)
    df_train.to_json(folder + train_name, orient='records', lines=True)
    df_test.to_json(folder + test_name, orient='records', lines=True)
randomly_split_dataset('finetune_data/', 'dataset.jsonl')

# An elaborate, high quality docstring for the above function:
"""

```

Sample response

""" Randomly split a dataset into train and test.

Parameters

folder : str

The folder where the dataset is located.

filename : str

The name of the dataset file.

split_ratio : list, optional

The ratio of train and test, by default [0.8, 0.2]

Returns

None

The function doesn't return anything, it just saves the train and test datasets in the given folder.

A sample of capabilities

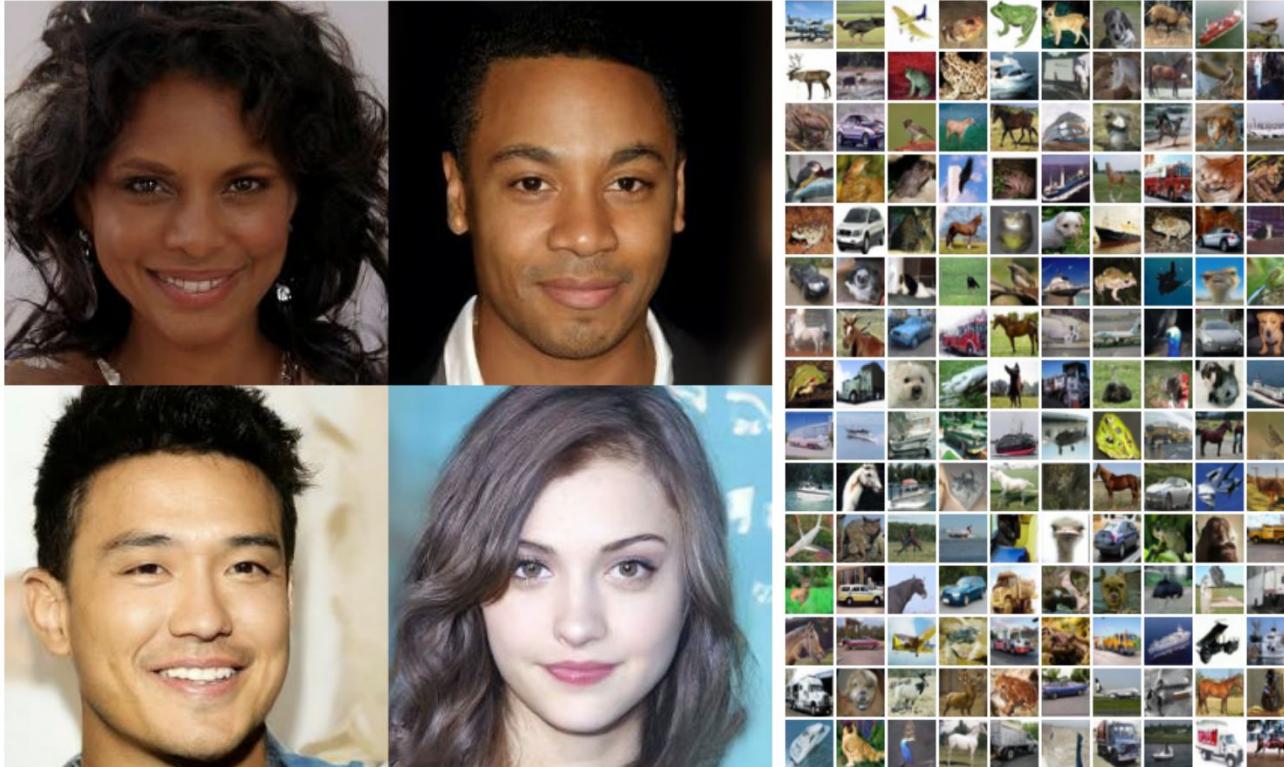


Figure 1: Generated samples on CelebA-HQ 256×256 (left) and unconditional CIFAR10 (right)

Units and layers

[Artificial] neuron (unit)

Linear unit: an aggregating function, typically a weighted sum of inputs:

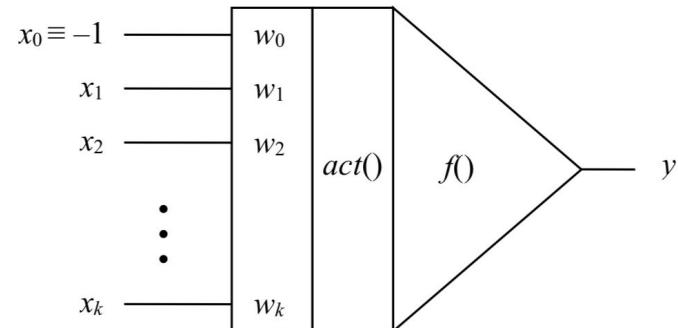
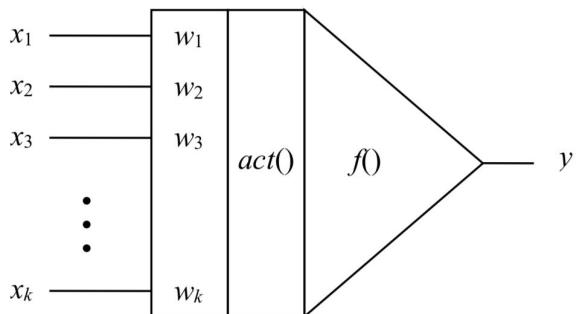
$$y = \sum_{i=1}^n w_i x_i + b$$

where w_i – weights, b – bias

Convenient reformulation:

$$y = \sum_{i=0}^n w_i x_i$$

where x_0 is assumed to be constant (typically 1 or -1).



Units: comments

- Weighted sum of inputs sometimes referred to as *excitation* or *activation*.
 - Warning: the latter is often conflated with the output of a unit (after applying activation function).
- Weights = parameters.
- Number of inputs = a hyperparameter.
- Typically implemented in a stateless (memory-less) manner.
- Implements a *dychotomizer*: divides the space of inputs into the positive and the negative half-space.

Implication of linearity

- There is no point in composing linear units: their composition is a linear unit (weighted sum of weighted sums).
- Nevertheless, occasionally used in contemporary architectures to reduce computational costs or for other purposes (to be shown later).

Interlude: other types of excitations

More precisely: other ways of defining unit's excitation.

Radial unit: activation determined by a radial function, e.g.

$$||x - w||_2$$

- Excitation captures the similarity of the input vector x to the weight vector w .
- Used in Radial Basis Function models (RBFs),
 - An architecture designed for supervised or unsupervised clustering.
- Relatively rare nowadays.

Nonlinear unit

The weighted sum of inputs ('activation') is passed through a form of nonlinearity:

$$y = \sigma(\sum_{i=0}^n w_i x_i)$$

where σ is an activation function.

Composition of nonlinear units is not redundant anymore (compared to linear units): it increases expressibility.

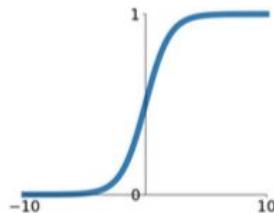
Types of nonlinearities (activation functions) used in ANNs:

- Bounded codomain: typically S-shaped, squeezing functions :
 - Bipolar: tanh, codomain [-1; 1]
 - Unipolar: sigmoid function, codomain [0; 1]
- Unbounded codomain: e.g., Rectified Linear Unit (ReLU): $\text{relu}(x) = \max(x; 0)$

Activation functions

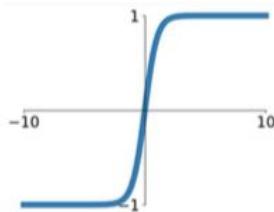
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



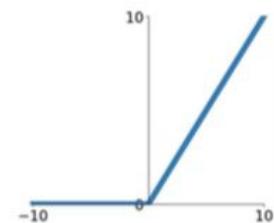
tanh

$$\tanh(x)$$



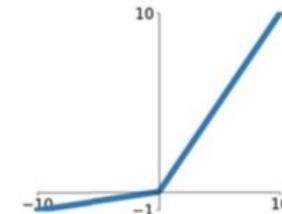
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

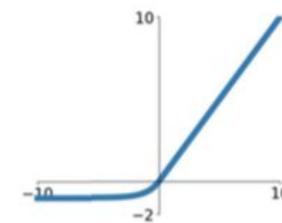


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



More about that later.

Layers

- In the ‘second spring’ of neural networks (late 1990s), the discourse was (still) conducted mostly on the level of individual units ([artificial] neurons).
- However:
 - NN and DL are paradigms for highly distributed, parallel information processing, so individual units are rarely critical.
 - An individual unit can be considered a degenerate layer (layer of size 1).
- Therefore, today we tend to talk in terms of layers.
 - Units in a layer typically operate in parallel and perform the same computation.
 - They may be differently parameterized, but don’t have to (see: weight sharing).Examples:
 - Dense layer: units have the same arity, but separate vectors of parameters.
 - Convolutional layer: units have the same arity (and also ‘shape’) and share parameters.

Layers

A natural abstraction of the concept of unit: a layer.

- Neurons typically combined into layers ('vectors' of units), because:
 - A single unit can produce only scalar output.
 - Vector outputs are often necessary.
- The simplest layers are one-dimensional, but sometimes we endow them with some higher-dimensional topology:
 - matrix (2D)
 - tensor (nD , $n > 2$)
- In practice, layer is the smallest unit of discourse in contemporary deep learning.
 - Note that a single unit can be considered a degenerate layer.

Layers vs. units

Convenient correspondence with linear algebra; for instance, a dense linear layer with n inputs (x) and m outputs (y) implements matrix multiplication:

$$y = Wx$$

The elements w_{ij} of the matrix is the j th weight of the i th unit in the layer.

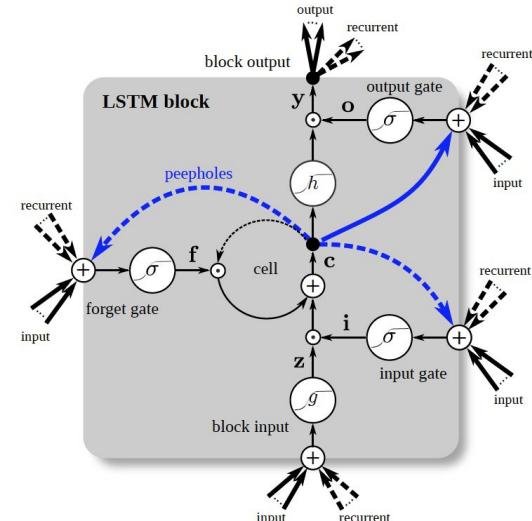
Layers vs. units

In most cases, a layer is just an ‘arrangement’ of units:

- Dense layer: a 1D ‘vector’ of units.
- Convolutional layer: a 2D ‘array’ of units.

However, some layers have more complex internal architecture:

- They encapsulate a smaller computation graph, and so hide complexity.
- A form of modularization.
- Example: Recurrent architectures, e.g. LSTM (Long Short-Term Memory unit/cell)



Computational capabilities of neural networks

Important features of the neural computing paradigm

- Continuous
 - Opens the door to the use of whole lot of maths (algebra, calculus, ...)
 - Does not preclude handling discrete variables (via, e.g., output thresholding)
 - Certain properties are formally provable (see next slide, for instance).
- Parallel or/and distributed, and hence:
 - Easy to deploy on parallel processing hardware like GPUs, TPUs, etc. (even in training phase; see *federated learning*).
 - Robust to local failures (kind of, mostly when implemented in hardware)
- Non-symbolic (sometimes referred to as subsymbolic)
 - However, neurosymbolic systems are becoming popular in recent years.

ANNs are universal approximators (Cybenko, 1989)

Let ϕ be a nonconstant, bounded, and monotonically-increasing continuous function. Let I_m denote the m -dimensional unit hypercube $[0,1]^m$, and $C(I_m)$ denote the space of continuous functions on I_m .

Then, given any $\epsilon > 0$ and any function $f \in C(I_m)$, there exists an integer N , real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \dots, N$, such that we may define

$$F(x) = \sum_{i=1}^N v_i \phi(w_i^T x + b_i)$$

as an approximate realization of the function f , where f is independent of ϕ ; that is,

$$|F(x) - f(x)| < \epsilon$$

for all $x \in I_m$.

ANNs are universal approximators (Cybenko, 1989)

Summary: a linear combination of monotonic nonlinearities (e.g., an ANN with one sigmoid layer followed by a linear layer) can approximate any bounded continuous function f arbitrarily well.

Practical implication:

- In principle, any function meeting the assumptions of the theorem can be approximated by a 2-layer network (linear combination of N nonlinear units).
- Approximation error is only function of N , and can be arbitrarily reduced by increasing the number of units.

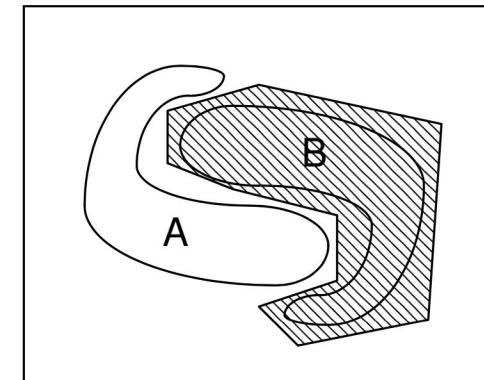
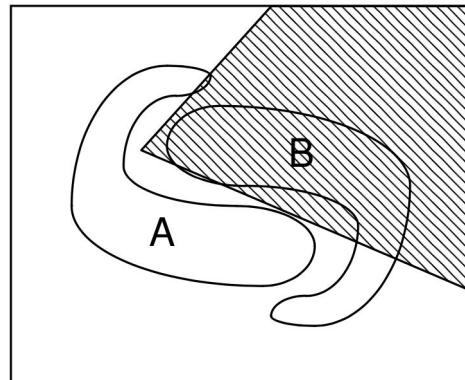
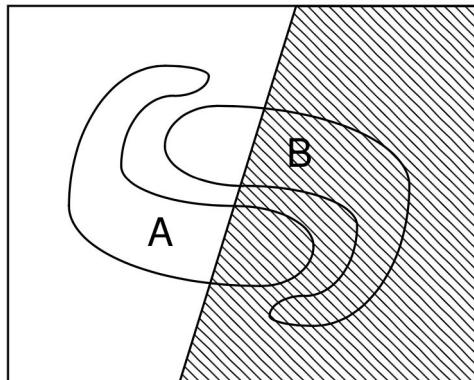
However:

- Cybenko's theorem confirms the *existence* of such approximation. *Finding N and $w_i s$* is an open problem (intractable in general, see later).
- In practice, f is not given – we know only a *sample* of its behavior, i.e. the training set.

A note on expressibility

Universal approximation with Cybenko's model may require unpractical number of units N. In practice, we boost the expressive capabilities of ANNs by increasing the number of layers. One of the possible/popular interpretations:

- A single unit partitions the input space into two half-spaces.
- Two layers can 'carve out' an arbitrary convex polyhedron in the input space.
- A third layer can combine multiple convex regions into an arbitrary shape.
- Single units may approximate logical operators: and, or, nor, ...



The need for composition of nonlinearities

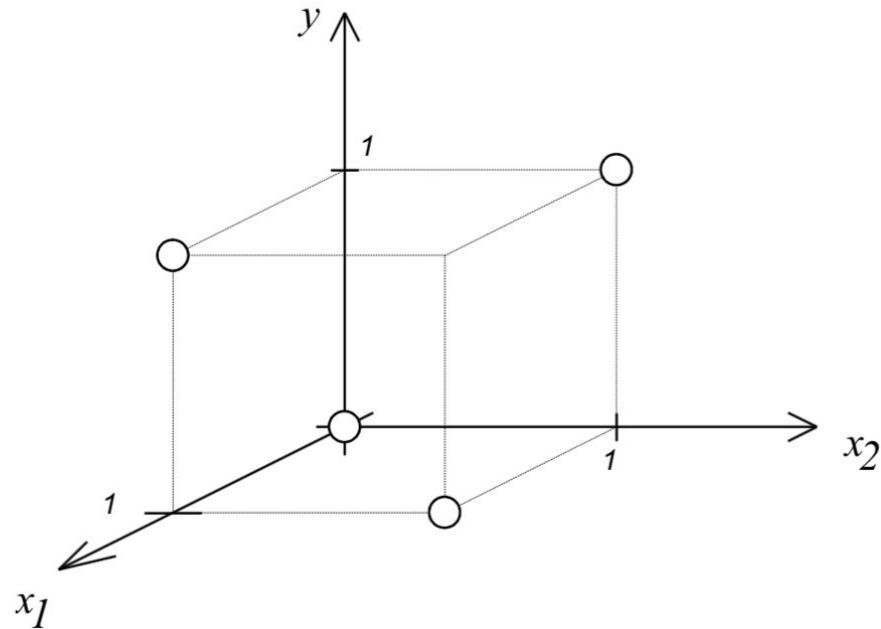
Even apparently simple problems are not linearly separable.

Example: the XOR problem.

Truth table (exhaustive ‘training set’):

x1	x2	y=XOR(x1,x2)
0	0	0
0	1	1
1	0	1
1	1	0

Desired behavior of the model
(in the classification sense, i.e.
producing the output with the correct
sign) cannot be realized with
a single layer.



Layers can be arranged in arbitrary architectures

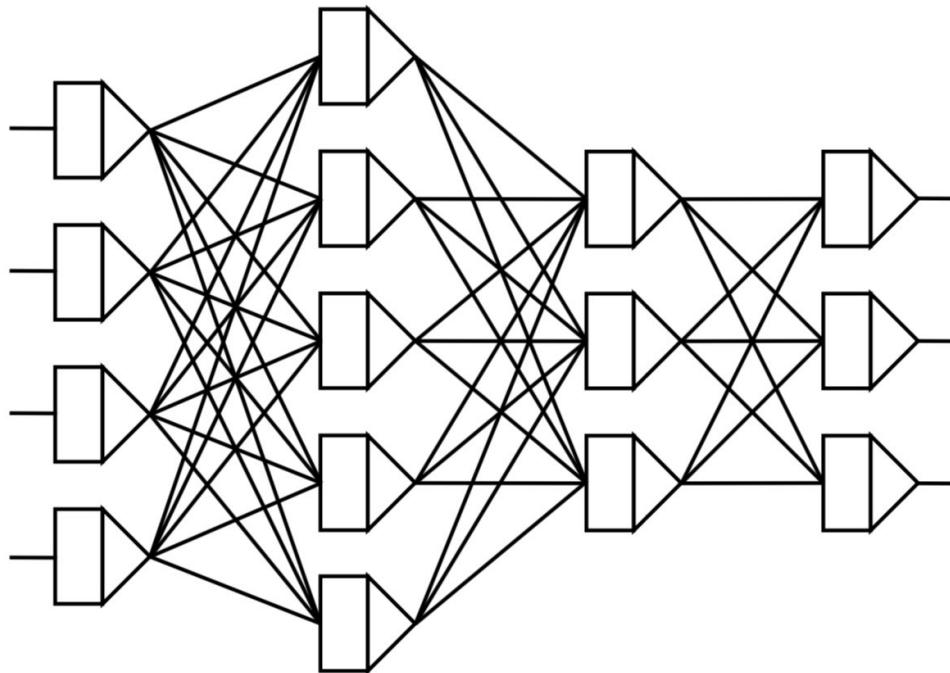
- Main line of division:
 - Feed-forwarded architectures (can be implemented in a stateless way)
 - Feed-back architectures (e.g. recurrent, but not only those)
- Implications:
 - The former are stateless, the latter stateful.
 - Both types used in contemporary models.
- Note: Some architectures do not feature explicitly defined inputs/outputs
 - E.g., the Hopfield network is a clique of units, each collecting signals from all units, including itself).

Next slides: A few glimpses on some of the architectures presented in this course.

Fully connected (FC), feed-forward architecture

Each unit from a given layer receives signals from all units in the previous layer.

- A composition of multiple dense layers
- No connections to preceding layers ('shortcuts').

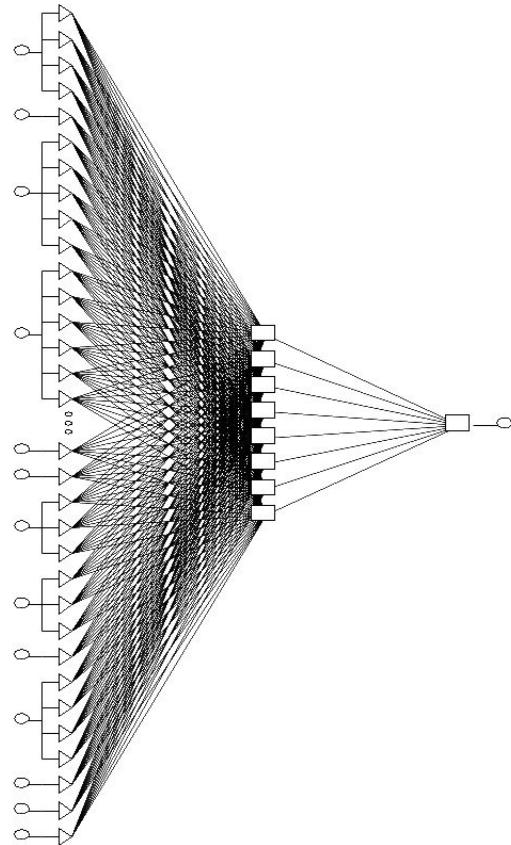


Interlude

What is the number of layers in this model?

Two different conventions:

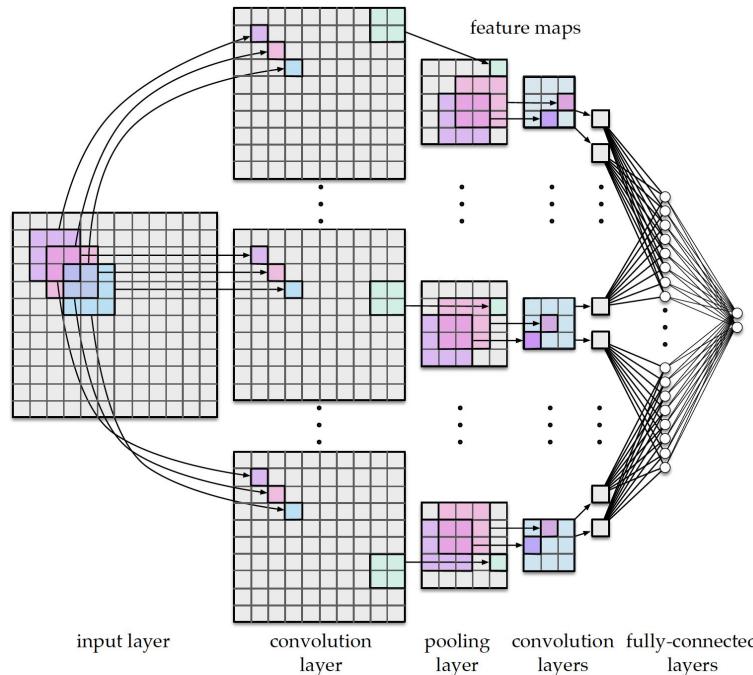
- Count only the nonlinear layers: 2 layers.
 - More common in scientific discourse.
- Count also the input ‘connectors’: 3 layers.
 - More common in software implementations.
 - E.g. ‘input layer’.
 - Input connectors are often implemented as a separate software component (object/class).



Mixed convolutional+FC architecture

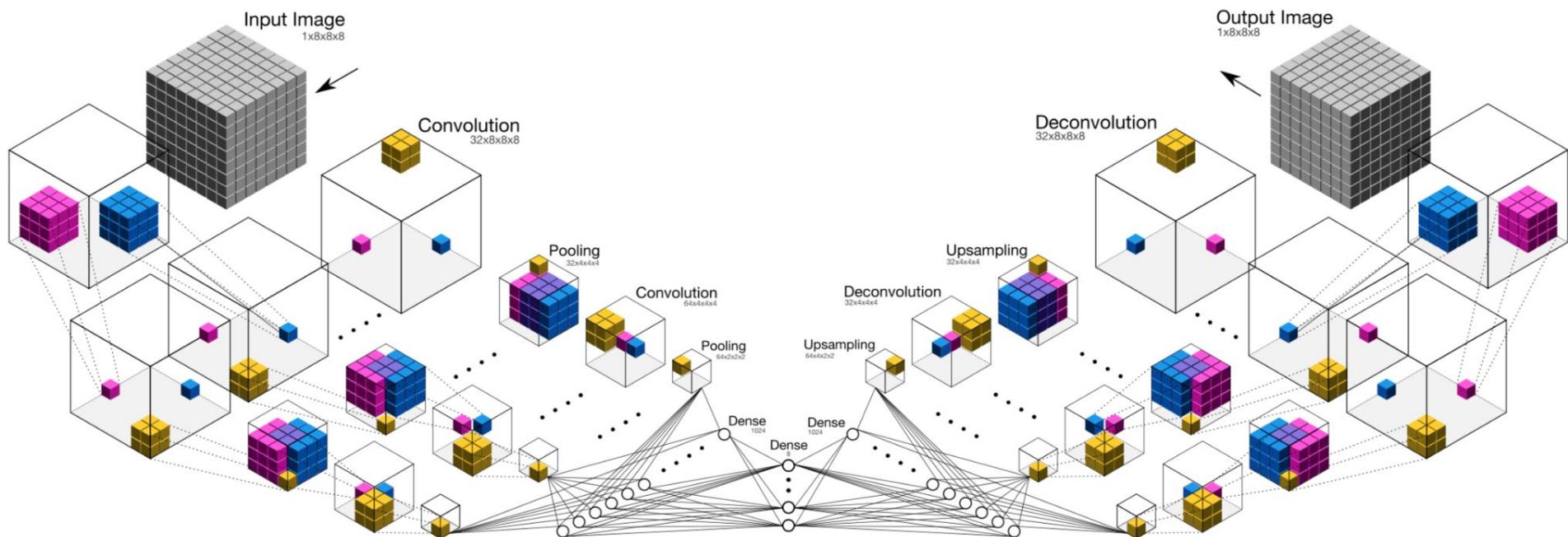
Fully connected, a.k.a. dense.

The first architecture of choice for problems of object/image recognition.



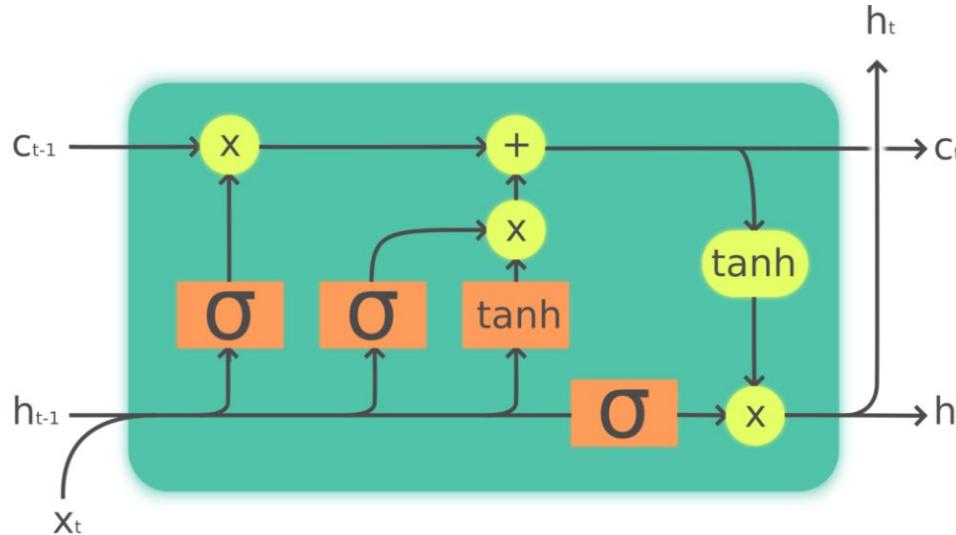
Convolutional autoencoder (3-dimensional)

- Formally a special case of ConvNet (convolutions + dense layers).
- Trained via autoassociation, i.e. reproduce the input.



Recurrent cell

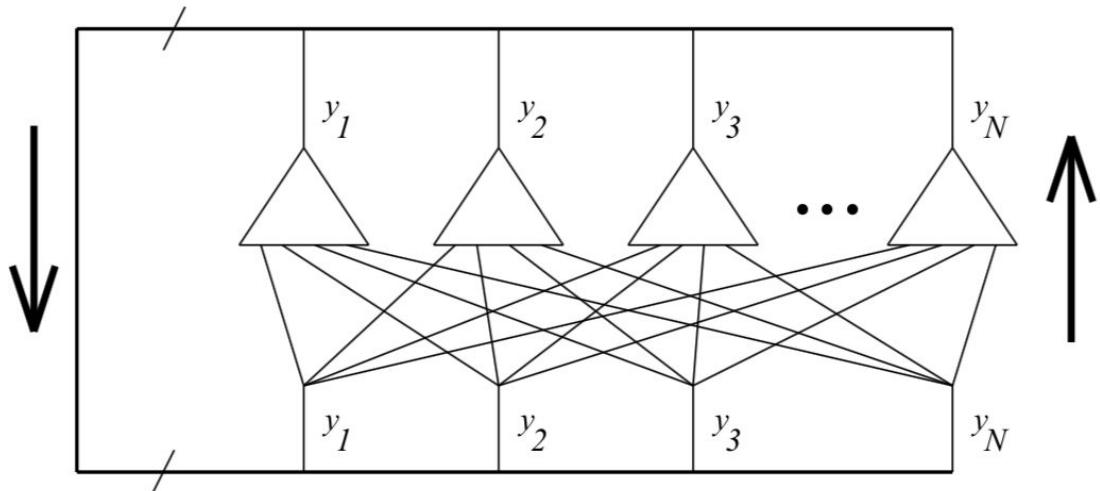
- Long short-term memory model (LSTM)
- Note: recurrence not explicitly visible in the diagram.
- The diagram shows only processing for a single time step.



Fully connected graph (clique) architecture

Each unit receives inputs from the outputs of all other units.

- Feedbacks possible
- Technically: a recurrent model (though mainly of theoretical importance)
- Notable example: Hopfield network



Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proceedings of the National Academy of Sciences. 79 (8): 2554–2558.

Training neural networks

Classes of learning tasks

1. ‘Contracting’ tasks: dimensionality m of the output (much) lower than dimensionality of input n: $n \gg m$

Subclasses:

- Classification: $\mathbb{R}^n \rightarrow \{1, \dots, m\}$
 - Binary classification: $\mathbb{R}^n \rightarrow \{0, 1\}$; alternatively: $\mathbb{R}^n \rightarrow \mathbb{B}$
 - Including anomaly detection (the anomaly class is very small and/or hard to model)
 - Special case: classification with missing inputs.
- Multi-label classification: predicting a number of labels at once: $\mathbb{R}^n \rightarrow \mathbb{B}^m$
 - Special case: structured prediction (structured output)
- Regression: $\mathbb{R}^n \rightarrow \mathbb{R}$
 - Density estimation: $\mathbb{R}^n \rightarrow [0, 1]$

Classes of learning tasks

2. Autoassociation tasks: $n = m$

Examples:

- Denoising: $\mathbb{R}^n \rightarrow \mathbb{R}^n$
- Imputation of missing values: $\mathbb{R}^n \rightarrow \mathbb{R}^n$

3. ‘Expanding’ tasks: $n \ll m$

Examples:

- Synthesis (generation) and sampling: $\mathbb{R}^n \rightarrow \mathbb{R}^m$

Classes of learning tasks

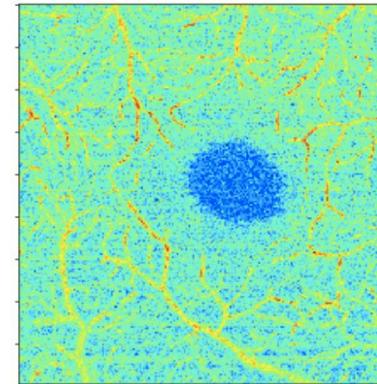
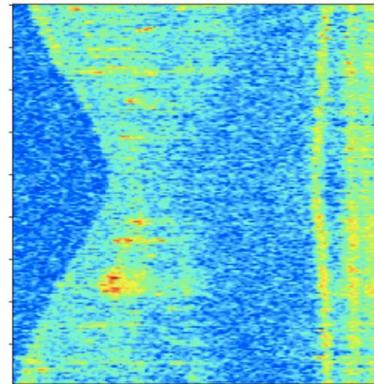
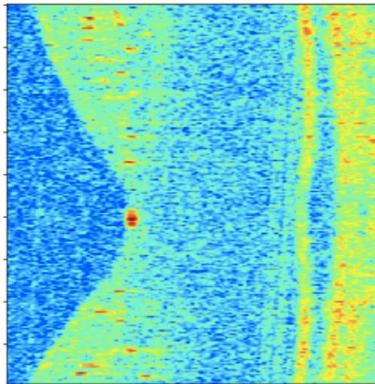
Tasks involving text, i.e. sequences of tokens/symbols:

- Transcription: $\mathbb{X} \rightarrow \mathbb{T}$, where \mathbb{T} is text (a sequence of words), and \mathbb{X} is a piece of ‘unstructured information’ (e.g., image; example: Google Street View reading house numbers).
- Machine translation: $\mathbb{T}_1 \rightarrow \mathbb{T}_2$

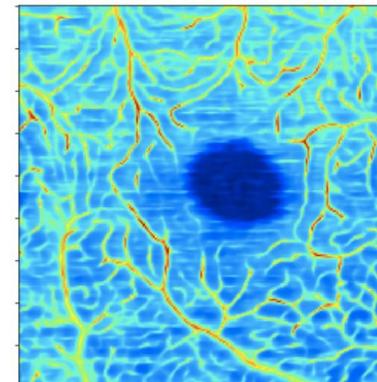
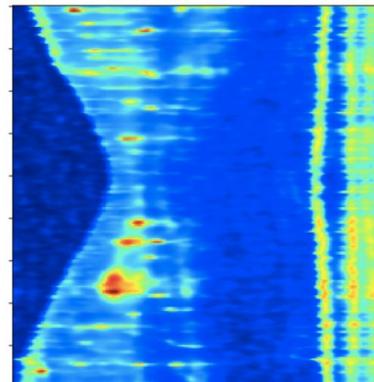
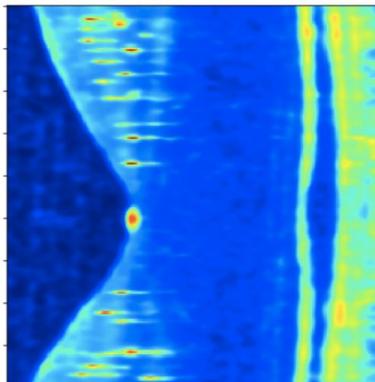
Immense advantage of DL: universality:

- all these tasks can be approached with DL,
- using a very similar toolbox of components: layers, cells, etc. – in general functions.

Example: Denoising OCT, convolutional autoencoder



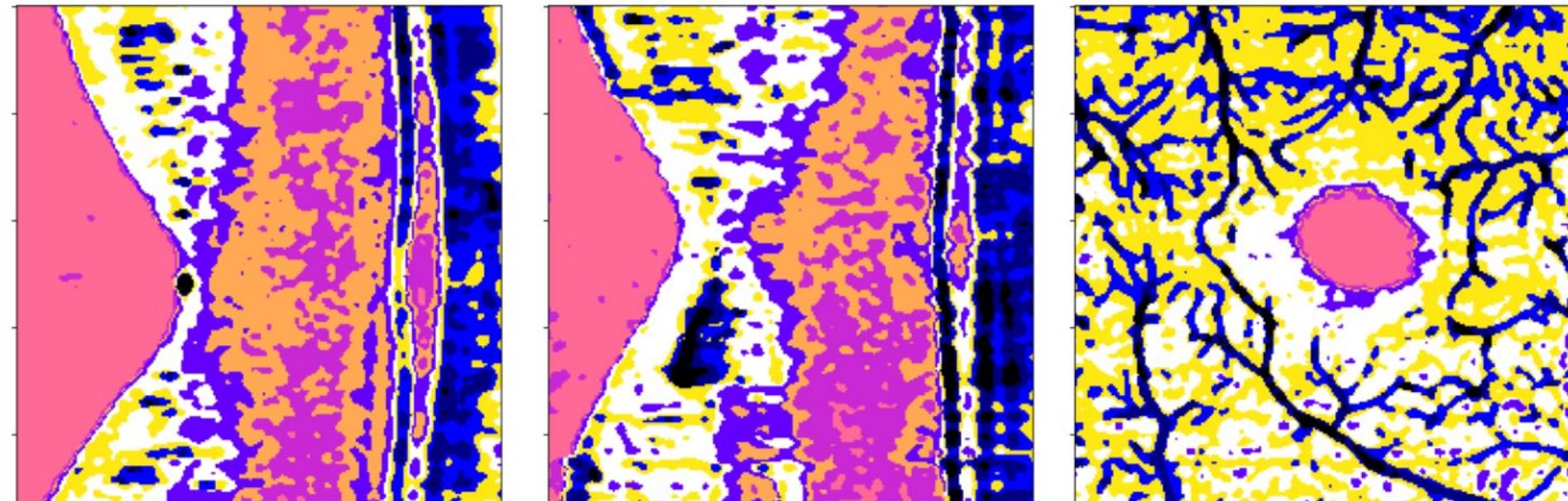
(a) Preprocessed OCT image Ω'_0 .



(b) Global reconstruction result $\hat{\Omega}_0$.

Example: Segmenting OCT

Convolutional autoencoder + cluster analysis in the latent space



(c) Segmentation result Π_0 of the Ω_0 image using **VAE4** model.

Training NNs

Q: If ANNs is a computing paradigm, how do we program them?

A: Via learning from examples.

We adopt the setting of machine learning (ML): given a sample of examples (*training set*), an inducer/learner produces a *function* (classifier, regression machine, etc) that [attempts to] minimize certain loss function.

Most ANN learning algorithms leave network topology intact, and optimize only networks' parameters.

Training NNs

We pose the problem of learning a NN as an optimization problem.

- Notable example from the past: Support Vector Machines.
- This is not a rule in ML: see, e.g. decision trees, decision rules, clustering, ...

This has several advantages:

- Sound formalization.
- Applicability of various optimization algorithms.

Classes of training algorithms:

- Direct: exist for specific, simple models:
 - Hopfield network: optimal weights are equal to correlations between input attributes,
 - Single sigmoid unit: implements logistic regression.
- Generate-and-test (e.g., evolutionary computation)
- Gradient-based: most popular.

NP-hardness of NN training

The task of training is NP-hard (Blum and Rivest, 1992):

We consider a 2-layer, 3-node, n -input neural network whose nodes compute linear threshold functions of their inputs. We show that it is NP-complete to decide whether there exist weights and thresholds for this network so that it produces output consistent with a given set of training examples. We extend the result to other simple networks.

Implication: All practical training algorithms are heuristics, i.e. do not guarantee producing an optimal solution (finding the global optimum of a loss function).

Loss functions

Properties:

- Defines the quantity to be minimized by the learner (network).
- Its minima correspond to weight configurations that cause the model to behave as desired.

The set of adequate loss functions is constrained by the type of task.

The main line of division:

- Regression tasks:
 - Unconstrained target
 - Unconstrained network output (usually linear activation function in the last layer)
 - Adequate loss functions: L1, L2, Huber, and similar
- Classification tasks:
 - Target interpreted as probability (typically 0 or 1)
 - Model's output interpreted as probabilities, and thus must be in [0,1]
 - Adequate loss functions: cross-entropy (binary or categorical)

Loss functions for regression tasks

Popular loss functions:

- Quadratic
- Absolute error
- Huber

Quadratic loss (L_2)

Quadratic loss (square of Euclidean norm):

$$L_2(\hat{y}, y) = \|\hat{y} - y\|_2^2$$

In practice same as Mean Square Error:

$$MSE(\hat{y}, y) = \frac{1}{m} L_2(\hat{y}, y)$$

where m is the dimensionality of the vectors.

Exact interpretation depends on the context, e.g.:

- m = number of training examples, or
- m = number of predicted variables (multivariate regression), or
- both.

Properties:

- Differentiable in entire domain.
- Penalization grows quadratically with deviation/divergence from the target.
- Gradient grows linearly with deviation.

Absolute loss (L_1)

Absolute loss:

$$L_1(\hat{y}, y) = \|\hat{y} - y\|_1$$

When averaged, called Mean Absolute Error (MAE).

Properties:

- Differentiable in entire domain, except at 0.
 - In practice, we assume gradient being equal to 1 or -1 at 0.
- Penalization grows linearly with deviation/divergence from the target.
- Gradient does not depend on deviation – it is constant (+1 or -1).
- Gradient changes abruptly at 0.
 - Implication: potential instability of gradient-based training algorithms.

Huber loss

Definition (by thresholding on the ordinate):

$$L_\delta(\hat{y}, y) = \begin{cases} \frac{1}{2}L_2(\hat{y}, y) & \text{if } L_1(\hat{y}, y) \leq \delta, \\ \delta(L_1(\hat{y}, y) - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

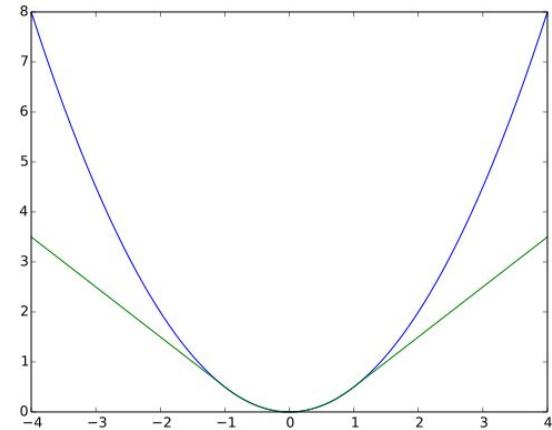
Properties:

- Combines
 - smooth characteristics close to 0
 - with fixed gradient further from it.

Sometimes called ‘Smooth L1’

- and defined in terms of abscissa, rather than the ordinate (the threshold of 1 could be set otherwise):

$$\text{smooth L}_1(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$



Data preprocessing for regression tasks

Inputs:

- Should be scaled to a small interval, possibly symmetric around 0.
 - Scaling to asymmetric intervals (e.g. [0,1]) impedes the performance of gradient-based algorithms (all weight corrections/deltas have the same sign for the first layer of the model).
 - However, this problem can be partially alleviated by the presence of momentum term (or other extensions of the basic SGD)
- Good practice: standardization: $(x - \mu_x)/\sigma_x$
- This advice holds for continuous input variables in all tasks, not only regression.

Output:

- A linear unit, to ensure that all values of the dependent variable can be produced.

Loss functions for classification tasks

Popular loss functions:

- Zero-one
- Cross-entropy (binary and categorical)

However: class label is a nominal (categorical) variable.

How to make model/network ‘communicate’ with such variables?

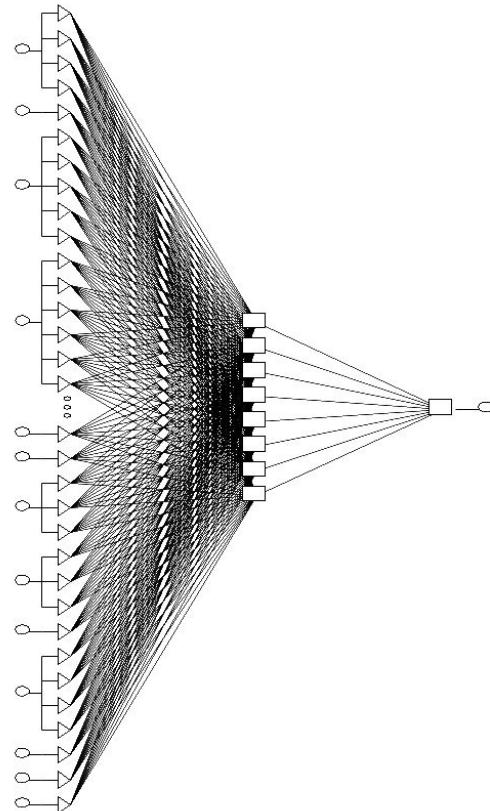
This leads to two questions:

1. How do we *present* categorical variables to the model?
2. How do we *adjust* the model to that presentation?

(see next slides for answers)

1. Presentation: One-hot encoding

- Given an n -valued categorical variable, we encode its k th value as an n -element vector, with all elements set to 0, except for the k th one, which is set to 1.
 - This prevents misleading the model about the existence of an apparent ordering of values.
 - (which would, for instance, happen when naively mapping the values to, say, the $[0, 1]$ interval)
- Essential for outputs.
 - Also highly recommended for inputs (though less critical); see the architecture on the right.



One-hot encoding: remarks

- Expresses a particular value of a categorical variable as a probability distribution (p_i s).
- No need for applying one-hot encoding for binary ($n=2$) categorical variables.
 - $p_2 = 1 - p_1$
 - One output is enough.
- One-hot encoding significantly increases the number of parameters of the model
 - Fortunately, the increase is linear w.r.t. the cardinality of the variable.
- More fancy encodings for categorical variables exist.
 - This is also related to embeddings (to be discussed later in this course).

2. Adjustment: Softmax activation function

Generalization of logistic function to multiple dimensions.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}} \quad \text{for } i = 1, \dots, n \text{ and } \mathbf{z} = (z_1, \dots, z_n) \in \mathbb{R}^n.$$

Maps the n -dimensional input to $(n-1)$ -dimensional simplex.

Invariant under translation by constant offset c :

$$\sigma(\mathbf{z} + \mathbf{c})_j = \frac{e^{z_j+c}}{\sum_{k=1}^K e^{z_k+c}} = \frac{e^{z_j} \cdot e^c}{\sum_{k=1}^K e^{z_k} \cdot e^c} = \sigma(\mathbf{z})_j.$$

Implications: only the *relative* differences between the elements of \mathbf{z} matter.

Note: invariance to *some* transformation was expected: this is how we drop that one degree of freedom ($n \rightarrow n-1$).

Technical: Often implemented as a layer (rather than an activation function).

Properties of softmax

- Guarantees outputs to sum up to 1.
- Differentiable.
- Smooth derivatives.
- Efficient calculation of derivatives, analogously to the logistic activation function (special case of sigmoid).
 - For sigmoid activation function f :

$$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x},$$

$$\frac{d}{dx} f(x) = \frac{e^x \cdot (1 + e^x) - e^x \cdot e^x}{(1 + e^x)^2} = \frac{e^x}{(1 + e^x)^2} = f(x)(1 - f(x))$$

- For softmax:

$$\frac{\partial}{\partial q_k} \sigma(\mathbf{q}, i) = \sigma(\mathbf{q}, i)(\delta_{ik} - \sigma(\mathbf{q}, k)).$$

where i and k are indices of the output dimension and input dimension, respectively.

Zero-one loss

Implementation depends on the interpretation of output

$$L_{0-1}(\hat{y}, c) = 1[\arg \max(\hat{y}) \neq c]$$

where $1[]$ is the Iverson bracket:

$$1[true] = 1, \quad 1>false] = 0$$

Properties:

- Assumes one-hot encoding of the output.
- Requires multidimensional output.
- Non-differentiable.
 - Implication: limited usefulness in real-world settings.

Cross-entropy

The number of bits/nats required to encode distribution P using the distribution of another variable Q:

$$L(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x)$$

Discrete case:

$$L(P, Q) = -\sum_i p_i \log q_i$$

Compare to entropy, i.e., the expected information content:

$$-\sum_i p_i \log p_i$$

Properties:

- Differentiable in (0,1)
- Requires both outputs and targets to be in [0,1].
 - This is usually ensured by using sigmoid activation function or softmax operator.

Categorical cross-entropy: analogous, but for multi-valued categorical variables.

Cross-entropy vs. classification tasks: summary

Types of classification tasks:

1. Binary classification: two mutually exclusive classes
 - a. Model has one output producing values in $[0,1]$. Use:
 - i. Sigmoid activation function in the output layer.
 - ii. Binary cross-entropy
2. Multi-class classification: more than two mutually exclusive classes, $m > 2$
 - a. Model has m outputs producing values in $[0,1]$, the sum of which should always be 1.
 - i. Softmax activation function in the output layer.
 - ii. Categorical cross-entropy
3. Multi-label classification: two or more non-exclusive classes, $m \geq 2$
 - a. Classes are in general unrelated, so this can be seen as m classification tasks.
 - b. Model has m outputs producing values in $[0,1]$
 - i. Sigmoid activation function in the output layer.
 - ii. Binary cross-entropy (on all m outputs)

Other types of loss functions

- Cosine distance/proximity
 - Recommended for high-dimensional spaces.
- Charbonier loss (similar to Huber)
- Hinge loss
- Kullback-Leibler Divergence
 - (will cover when talking about variational models)

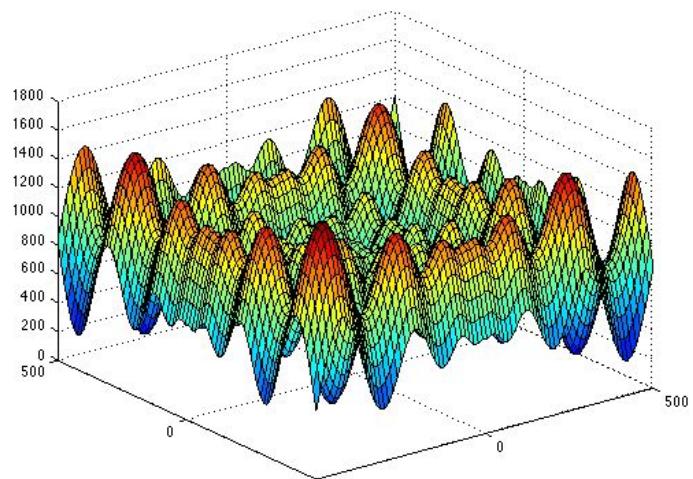
Remarks:

- More complex loss functions often built as aggregates of the ‘fundamental’ ones (weighted sum).
- In principle, any differentiable scalar-valued function can be used as a loss function.
 - E.g. structural similarity metric (SSIM) for images.
 - Caveats: computational cost, numeric stability.

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

Error/loss landscape

- Given
 - a fixed input x (and thus output y), and
 - fixed desired output y_{hat} ,
- the loss L is a function of model parameters and may be visualized as an “error landscape”.
- The difficulty of minimizing L depends on:
 - The existence of global optimum.
 - The number of local optima.
 - The number of saddle points.
 - The ‘ruggedness’.



Perfect ≠ good enough

Finding a good local optimum is often sufficient/satisfactory, because:

- Training data is usually limited anyway
 - The exact value of L is not known.
 - We only estimate it from the training sample.
- We often estimate L from an even smaller sample – a *batch*.
- There's often noise (or other artefacts) present in the data.
 - The observed values of variables (available in the training data) are different from the true ones, and thus our measurement of loss is imprecise.
- There exists a level of required performance that is 'good enough' (e.g., accuracy of classification).
 - See also: the law of diminishing returns.

Nevertheless, the main objective remains valid: minimize the loss, with the above in mind.

Gradient descent
(descent along gradient)

Gradient

Generalization of derivative to functions of multiple arguments:

$$\nabla_w L = \left[\frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_n} \right]$$

- A function ($\nabla_w L(x)$ is a specific value).
- $\nabla_w L(x)$ is a vector.
- Informs about the f 's rate of change w.r.t. its arguments.
- Applied to the loss function. L depends on
 - Model's parameters: w_i
 - Model's inputs: x_i
- Gradient is typically (in training) taken with respect to parameters (∇_w)
 - Occasionally calculated with respect to inputs (∇_x), e.g. in sensitivity analysis.
- Implication: diffability of components is essential in DL:
 - With respect to parameters: in order to train a given component.
 - With respect to inputs: in order to back-propagate the gradient to the components located earlier in the computation graph/chain (chain rule).

Gradient descent (GD)

A heuristic algorithm that uses gradient as a guidance.

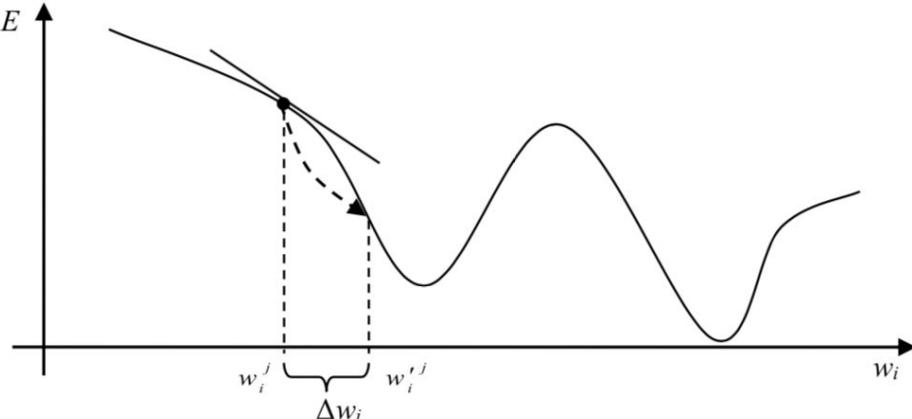
Basic formulation:

$$w \leftarrow w - \eta \nabla_w L$$

where:

- η – learning rate (a small positive constant).
- L is (in principle) calculated from all training data.

Intuition: follow the downward slope.



Gradient estimation

The exact value of the gradient is unknown!

- Requires complete knowledge of data distribution; we have only a sample (training set).

$$w \leftarrow w - \eta \nabla_w L$$

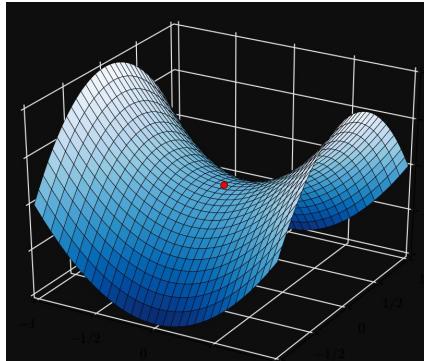
In practice, gradient can be estimated:

- from the entire training sample (averaged) $w \leftarrow w - \eta \nabla_w L(x)$
 - Downside: computationally expensive.
- from a single training example
 - Downside: risk of erratic behavior: examples can vary arbitrary, so gradient vectors point in different directions for consecutive training examples.
- from a batch (minibatch) of examples (the most common approach)
 - A compromise between the two above.
 - Batch size: a hyperparameter of the training process.
 - Batches are (typically) drawn “on the fly”, hence Stochastic Gradient Descent (SGD)

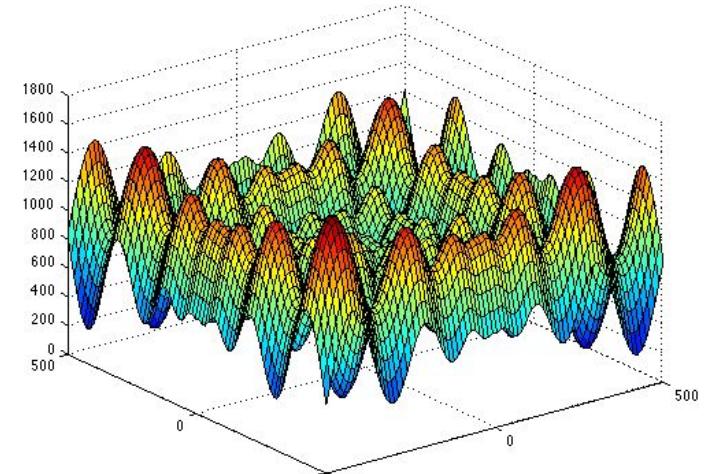
Note: batch size is not a purely technical parameter; it impacts the results.

Gradient descent: essential properties

- Guaranteed to converge to the global optimum for loss functions that are convex w.r.t. model's parameters (e.g. quadratic loss)
 - No such guarantees for other function classes.
- In DL the loss depends on w in a complex way, intermediated by a (potentially large) number of nonlinearities.
 - In practice, loss is never convex/unimodal w.r.t. to model parameters.
- Problematic cases ('loss of gradient'):
 - Local optima
 - Saddle points



*Not to be confused with vanishing gradient.



Gradient descent

Assumption: knowledge about gradient in the current location can be translated into parameter updates that take one closer to the global optimum.

- Does not hold in general, but works surprisingly well in practice.
- The probability of any given point being a local optimum quickly (exponentially) vanishes with the number of dimensions (model parameters).
 - This can be shown by analyzing the Hessian matrix of the loss function (the positivity of eigenvalues)
 - However, for the same reason, saddle points are more likely than local optima (DLB* 8.2.3; [Dauphin et al. 2014])
- Practical upshot: Gradient works well in practice,
 - ... although it's not obvious that following it (in an iterative search process) is always the best thing to do.

Gradient descent for a single linear unit: ‘delta rule’

Application/implementation of GD to a single linear unit trained under quadratic loss.

$$w \leftarrow w - \eta \nabla_w L$$

where $L = L_2$:

$$L_2(\hat{y}, y) = \|\hat{y} - y\|_2^2 = \sum_j (\hat{y}_j - y_j)^2 = \sum_j \delta_j^2$$

where j iterates over examples, and δ_j is the (signed) error committed on j th example.

$$\nabla_w L = [\dots, \frac{\partial L}{\partial w_i}, \dots]$$

For the i th weight:

$$w_i \leftarrow w_i - \eta \delta_j x_i$$

More generally:

$$w \leftarrow w - \eta \delta x$$

Delta rule for nonlinear units

Recall nonlinear unit:

$$y = \sigma(\sum_{i=0}^n w_i x_i) = \sigma(y')$$

By the chain rule:

$$\frac{\partial y}{\partial w_i} = \frac{\partial \sigma}{\partial y'} \frac{\partial y'}{\partial w_i}$$

The update rule is:

$$w \leftarrow w - \eta \delta x \frac{\partial \sigma}{\partial y'}$$

where the last term is the derivative of the activation function wrt ‘excitation’ y.

Error backpropagation

- Error backpropagation is essentially application of gradient descent to entire ANNs (rather than to individual units).
- It naturally follows from the chain rule.

Consider two sigmoidal units stacked upon one another (two layers).

- Assume for simplicity that both have just one input each.

$$\begin{aligned}y &= \sigma(\sum_{i=0}^n w_i x_i) = \sigma(w_0 x_0) = \sigma(y') \\&= \sigma(w_0 \sigma(w'_0 x'_0)) = \sigma(w_0 \sigma(y''))\end{aligned}$$

What is the gradient of $L(y)$ with respect to w'_0 ?

$$\frac{\partial L}{\partial w'_0} = \frac{\partial L}{\partial y} \frac{\partial \sigma}{\partial y'} \frac{\partial y'}{\partial x_0} \frac{\partial \sigma}{\partial y''} \frac{\partial y''}{\partial w'_0}$$

$$\frac{\partial L}{\partial w'_0} = \frac{\partial L}{\partial y} \frac{\partial \sigma}{\partial y'} w_0 \frac{\partial \sigma}{\partial y''} x'_0$$

Implications of the chain rule

$$\frac{\partial L}{\partial w'_0} = \frac{\partial L}{\partial y} \frac{\partial \sigma}{\partial y'} w_0 \frac{\partial \sigma}{\partial y''} x'_0$$

The effective updates reaching individual parameters (here w'_0) depend on:

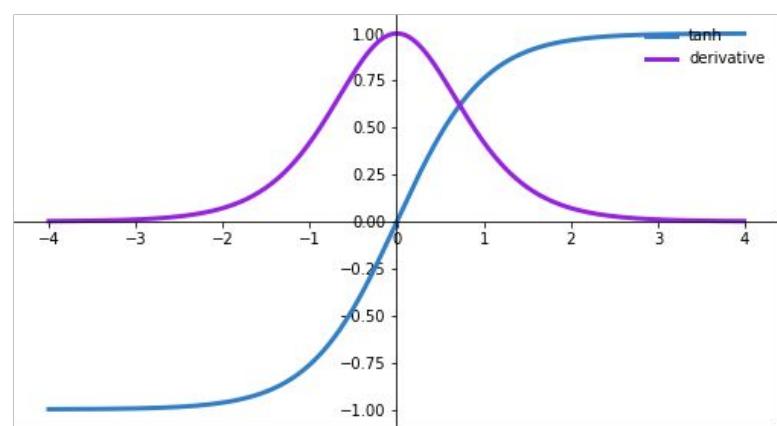
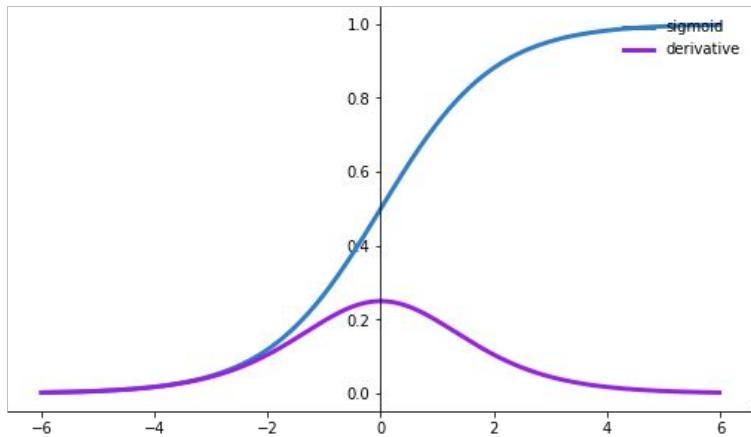
- The gradient of the loss function w.r.t. to the output y of the model
- The derivatives of activation functions
- The parameters (weights) of intermediate layers
 - What happens when w_0 is close to 0?
- The signal provided by the parameter (here: x'_0)

Note:

- In DL models, chains can have hundreds of terms.
 - A typical layer will add two elements to the chain.
- Updates likely to be smaller for layers further from the loss function (and thus from model's outputs) \Rightarrow *vanishing gradient problem*.

Derivatives of activation functions

The severity of the vanishing gradient problem depends on the characteristics of activation functions. S-shaped functions tend to make it more severe:



Most non-squeezing activation functions (e.g. ReLU) have non-zero gradient in large parts of their domains, and thus attenuate this problem.

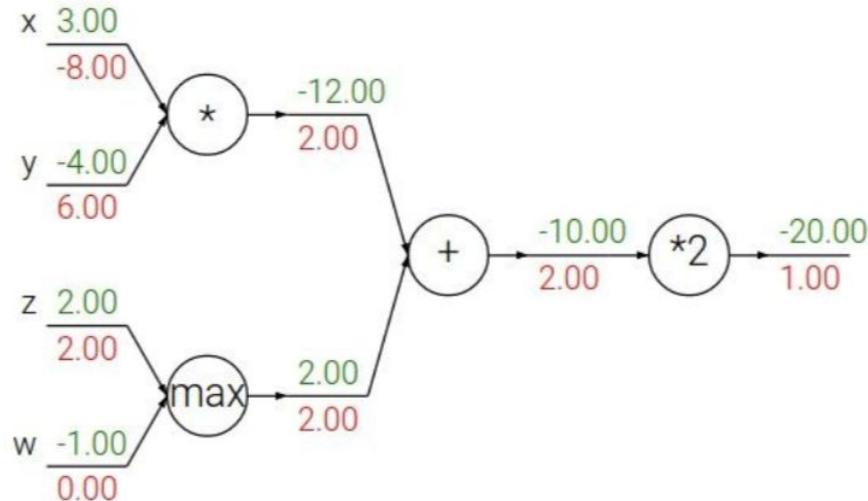
Models as computation graphs

Chain rule works for any differentiable components, not just scalar product and common activation functions.

- green: forward signal propagation
- blue: backward gradient flow (chain rule)

Note:

- The model is ‘blended’ with its loss function.
The optimizer treats them together as the function to be optimized.
- No significant difference between model inputs and model parameters (other than what the gradient is calculated with respect to).



Gradient descent: remarks

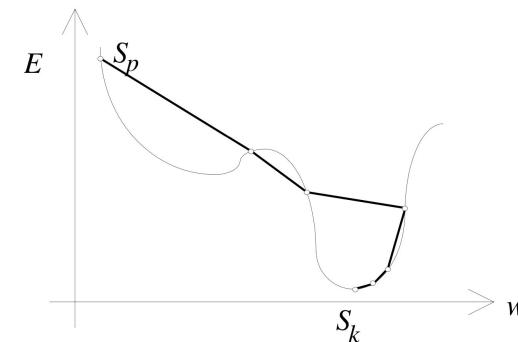
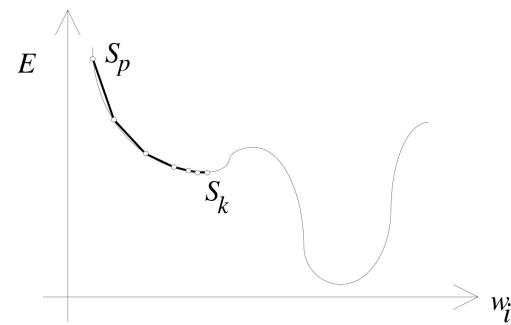
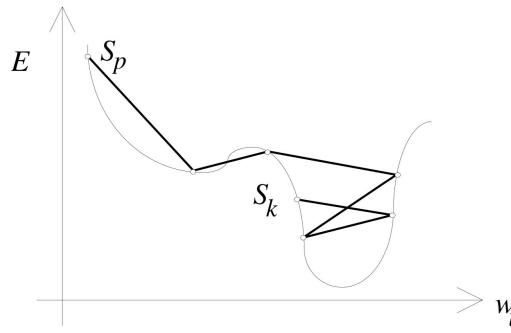
Contemporary training algorithms extend the GD paradigm with various additional mechanisms:

- Momentum
 - Nesterov momentum
- Adjusting learning rate dynamically,
 - sometimes independently for each parameter of the model.
- Backtracking
- ...

Setting the learning rate

The learning rate η modulates the impact of gradient on parameter updates.

- Hard to set in general; see figures below:
 - Left: too large η allows reaching the basin of minimum quickly, but makes it harder to hit the minimum precisely.
 - Middle: too small η increases the risk of getting stuck at a poor-quality local minimum or saddle point.
 - Right: well-tuned η (e.g. changed gradually during training) should provide good convergence (exploration) and precise localization of the minimum (exploitation).
- In practice, too large η is more harmful than too small (instability).



Adaptive tuning of the learning rate η

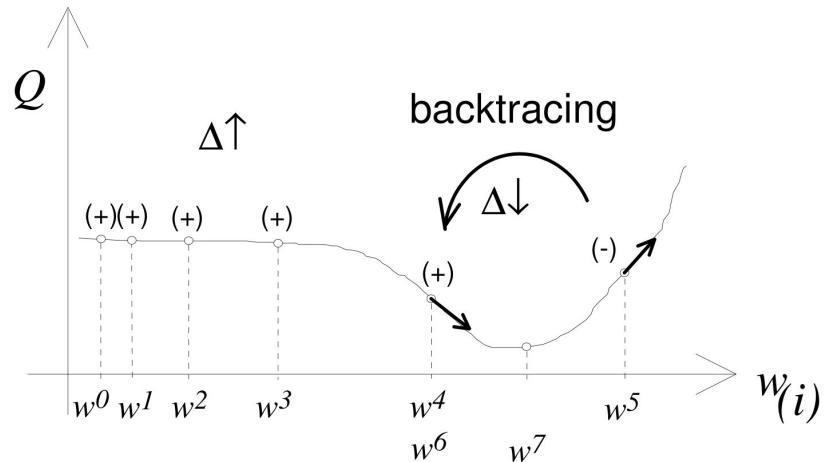
Some algorithms monitor loss and adjust η accordingly.

Resilient Backpropagation (Rprop*) monitors the sign of gradient (each dimension separately) from one epoch/batch to another:

- If it does not change, η is increased
- Change of sign signals passing a local minimum (on that dimension).

Algorithm's reaction:

- Restore the previous search point (backtracking)
- Decrease η



Implementing gradient estimation

Numerical differentiation

- Makes no sense in DL, as we know the analytical formulas of the expressions being differentiated.

Symbolic differentiation

- Elegant, but may lead to convoluted expressions that are costly to evaluate.
- See, e.g. SymPy <https://www.sympy.org/>



Automatic differentiation (autodiff)

- Relies on the concept of dual numbers
 - An analog to complex numbers, with the dual unit $\varepsilon^2 = 0$
- Covered in later parts of the course.

Run code block in SymPy Live

```
>>> diff(cos(x), x)
-sin(x)
>>> diff(exp(x**2), x)
  2
  (x )
2·x·e
```

End of module

Module 2: Generative Adversarial Networks

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2019–2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Outline

1. Learning generators for complex probability distributions: [Generative Adversarial Networks](#)
2. Conditional GANs for translating between domains: [Pix2pix \(PatchGAN\)](#)
3. Learning translation from unpaired examples [CycleGAN](#)
4. Making GANs more specialized for raster images: [Deep Convolutional GAN \(DCGAN\)](#)
5. Facilitating training of GANs:
 - o [Coulomb GANs](#)
 - o [Wasserstein GAN](#)

Generative Adversarial Networks

Generative Adversarial Nets

Generative Adversarial Networks

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio

<http://arxiv.org/abs/1406.2661>

Discrimination vs. generation

- Discriminative/predictive model: implements a mapping from an input space X to some output space Y .
Examples: classification, regression, ...
- Generative model: generates samples from a probability distribution Z to some output space X .
Typically, X has many dimensions (in contrast to the discriminative setting).
Example approaches:
 - Deep Belief Networks,
 - Boltzmann Machines,
 - Deep Boltzmann Machines,
- Key question: How to evaluate the adequacy of the samples generated by a generative model?

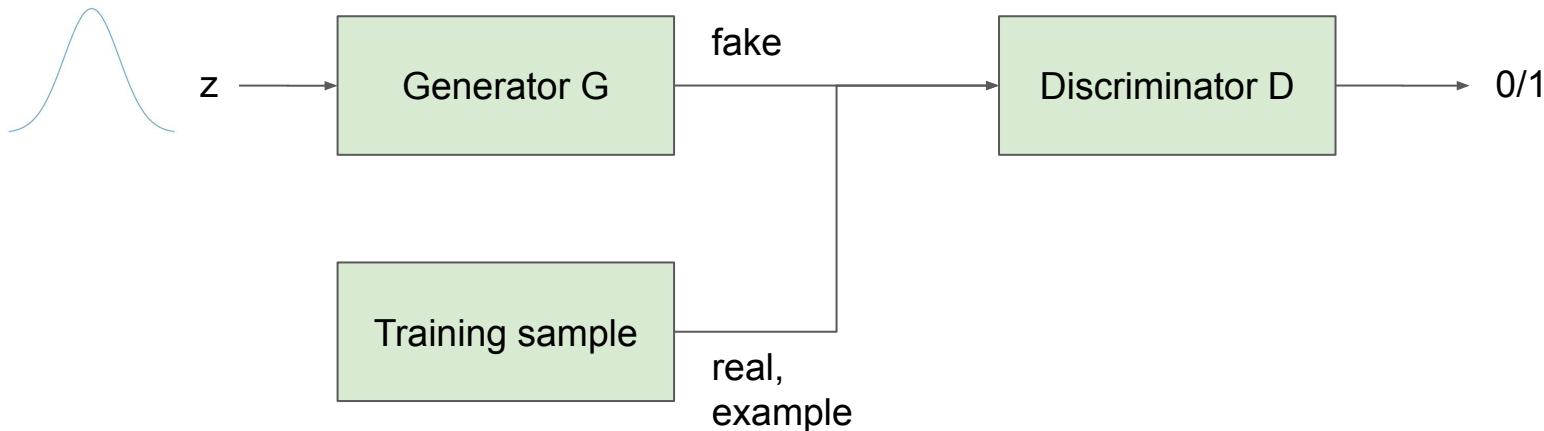
Adversarial setting

The goal: learn the distribution over some (arbitrary) data x , so that then sampling from that distribution is easy.

Approach:

- Define a prior on latent noise variables $p(z)$ (typically Gaussian $N_n(0,1)$)
- Represent a mapping from z to data space as $G(z)$, where G is a parameterized differentiable function.
- Define a parametric and differentiable discriminating model $D(x)$ that outputs a single scalar that represents the probability that x came from the distribution (training data).
- Train simultaneously:
 - D to maximize the probability of assigning the correct label to both training examples and samples from G .
 - G to minimize $\log(1 - D(G(z)))$

Adversarial setting



- Discriminator receives a fake example or a real example (they are not combined)
- Discriminator learns a binary classification task.
- As G and D are both differentiable, the gradient can flow from D to G, i.e. D can teach G.

Adversarial setting

Loss function defines a two-player minimax game with value function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})} [\log(1 - D(G(\mathbf{z})))].$$

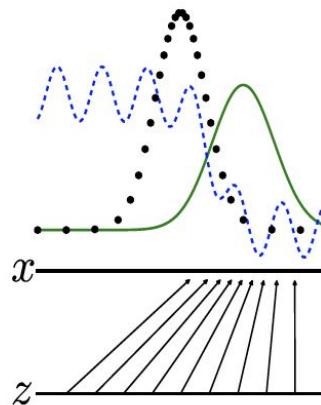
Simultaneously rewarding:

- D for maximizing its output for true data,
- G for causing D to fail to discriminate between the true data and the ‘fake’ examples it generates.

Illustration on 1D case

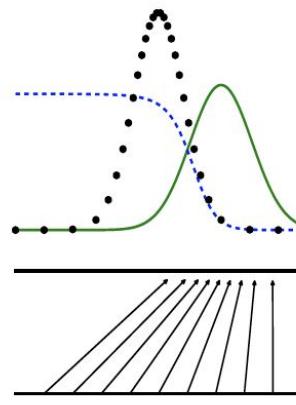
Latents z , fakes and reals are just scalars (or 1D vectors):

- blue, dashed line: decision boundary proposed by the discriminator
- black, dotted line: data distribution p_{data} (reals)
- green, solid line: generative distribution (fakes)



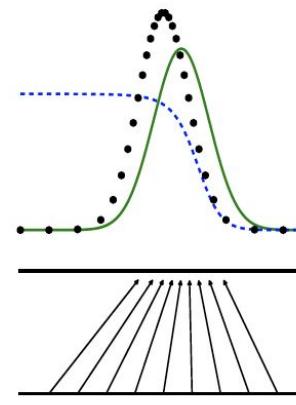
(a)

initial state



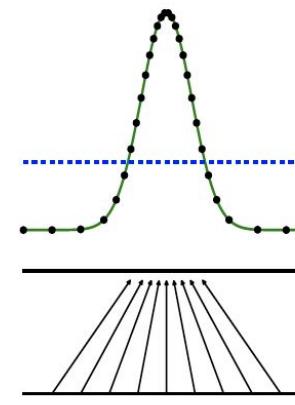
(b)

D trained a bit



(c)

G trained a bit



(d)

desirable final state

The complete training algorithm

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by **ascending** its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

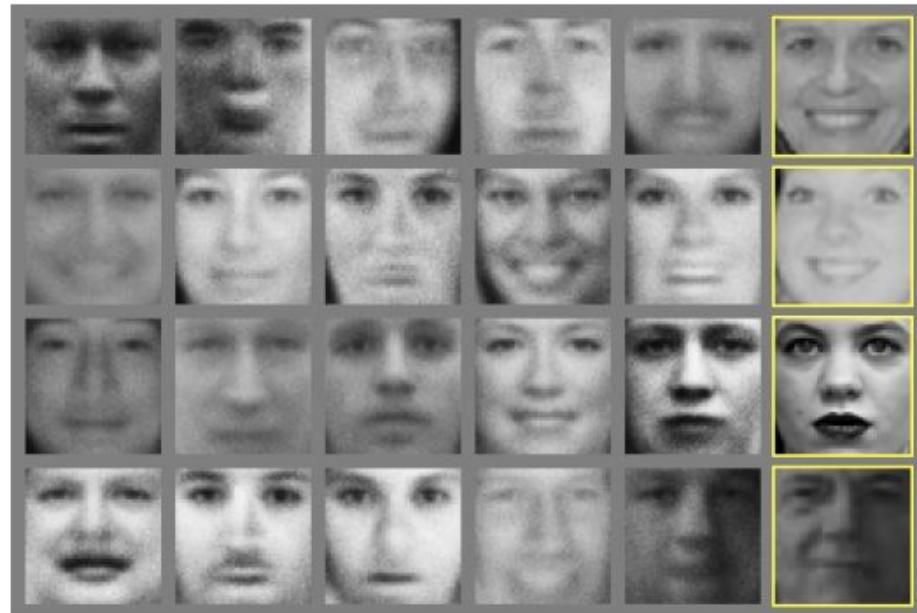
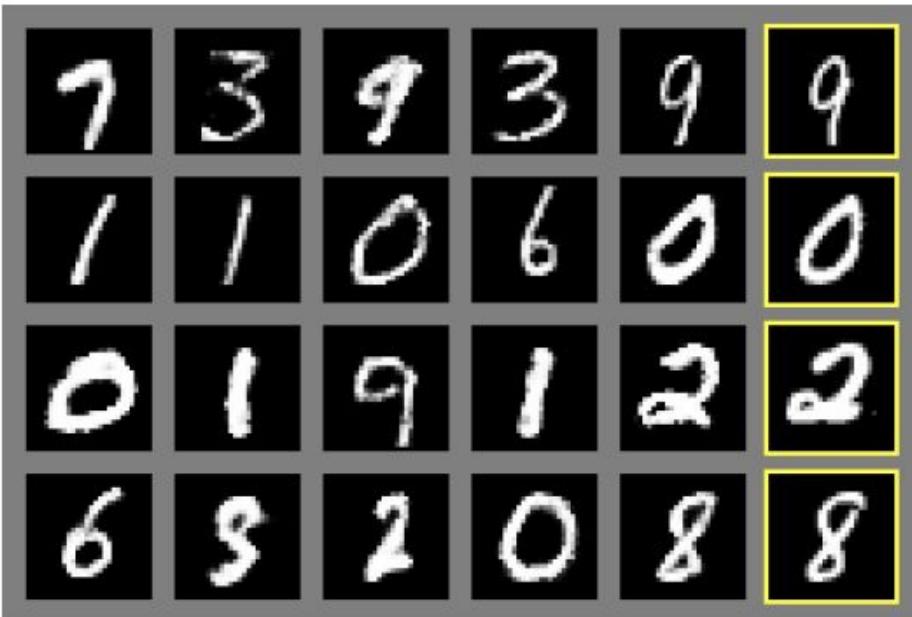
- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Visualization (MNIST, TFD = Toronto Faces Dataset)



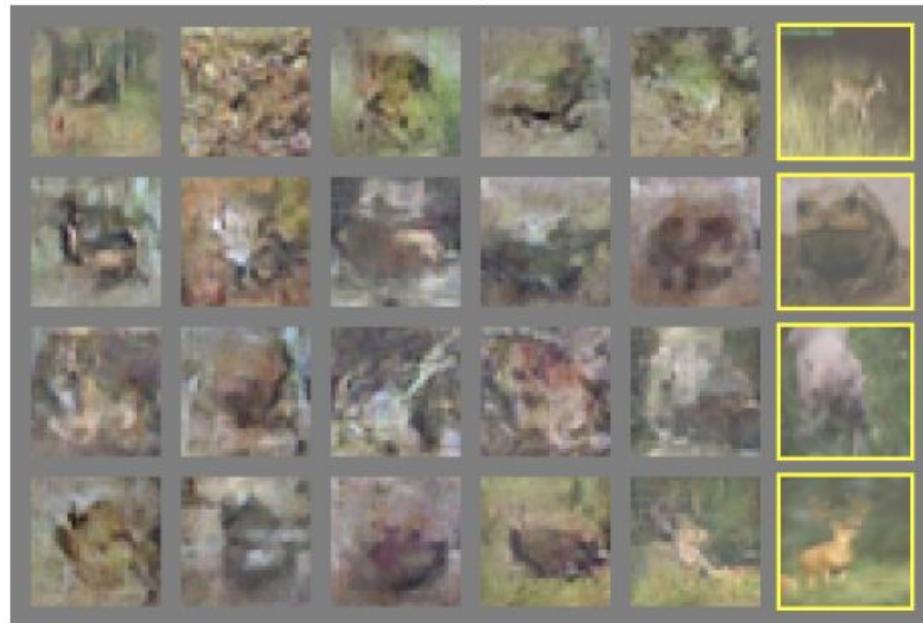
- Truly random sample, not cherry-picked.
- Rightmost column shows the nearest training example of the neighboring sample

Visualizations (CIFAR-10)



c)

Fully connected (left) vs. convolutional (right)



d)

Comments

- Notice the relationship to autoencoders.
 - In particular, G acts in a similar way to a decoder in autoencoders.
- How is it possible for G to generate such ‘rich’ outputs in response to low-dimensional and simple distribution?
- The paper contains also theoretical part - proofs of convergence.
 - However, under strong assumption, e.g. if at each step of Algorithm 1, D is allowed to reach its optimum given G
- The space of z values may have meaningful interpretation:
E.g.: The digits obtained by linearly interpolating between coordinates in z space and querying G :



Conceptual confrontation with other approaches

	Deep directed graphical models	Deep undirected graphical models	Generative autoencoders	Adversarial models
Training	Inference needed during training.	Inference needed during training. MCMC needed to approximate partition function gradient.	Enforced tradeoff between mixing and power of reconstruction generation	Synchronizing the discriminator with the generator. Helvetica.
Inference	Learned approximate inference	Variational inference	MCMC-based inference	Learned approximate inference
Sampling	No difficulties	Requires Markov chain	Requires Markov chain	No difficulties
Evaluating $p(x)$	Intractable, may be approximated with AIS	Intractable, may be approximated with AIS	Not explicitly represented, may be approximated with Parzen density estimation	Not explicitly represented, may be approximated with Parzen density estimation
Model design	Nearly all models incur extreme difficulty	Careful design needed to ensure multiple properties	Any differentiable function is theoretically permitted	Any differentiable function is theoretically permitted

Some notes

More general thoughts:

- When backpropagating gradient, D ‘translates’ binary labels and scalar classification loss (fake/real) into a multidimensional training signal for G.
 - The training signal G receives from D is very rich: gradient ‘flows’ (is defined for) through each pixel in the output tensor produced by G.
- Adversarial setting can be likened to competitive coevolution, a variant of evolutionary computation:
 - Typically two populations maintained in parallel, e.g. *predators* and *preys*.
 - For evaluation, a pair of predator and prey is drawn from the population, and they enter an *interaction* (e.g. play a game).
 - The outcome of the game determines (or contributes to) the *fitness* of both players.
 - This scheme has been successfully used to evolve game-playing strategies for multiple games and environments (e.g. checkers, Othello/reversi, and more).

On controlling generators

- Generators obtained with GANs approximate the target probability distribution, but cannot be effectively controlled.
 - ... other than by traversing the latent space and observing the output (see previous slides).
 - Nevertheless: the aspects of the distribution in X that tend to be independent will be usually linked by G with independent dimensions in Z , due to the isotropism of the normal distribution.
- In practice, training GANs can be tricky
 - May require lots of hyperparameter tuning.
 - Various forms of ‘pathologies’ are possible:
 - When D is too good, it’s hard for G to fool it (without temporarily deteriorating)
 - When G is too good, it’s hard for D to detect fakes (without temporarily deteriorating)
 - Formally: there are multiple Nash equilibria in the game, but only some of the are interesting.
- Can we extend a GAN so that it can be more explicitly controlled?

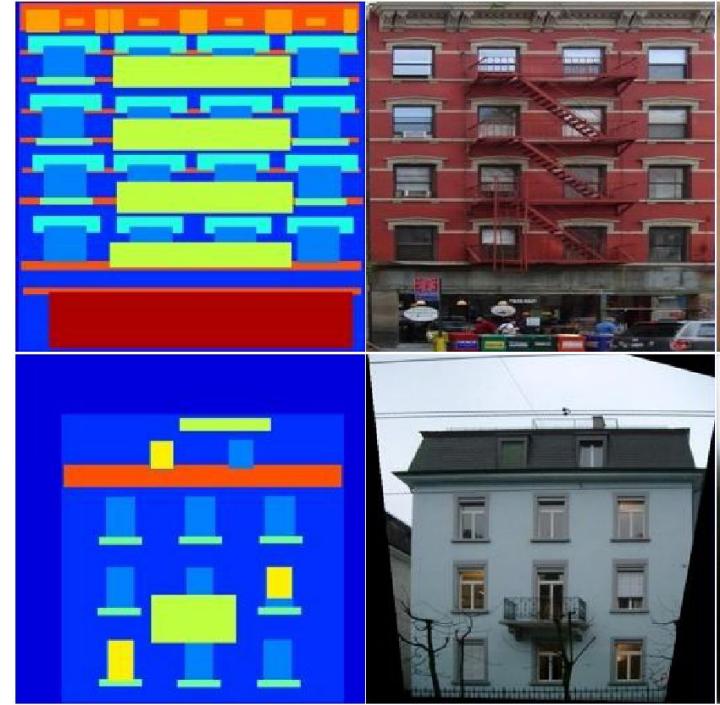
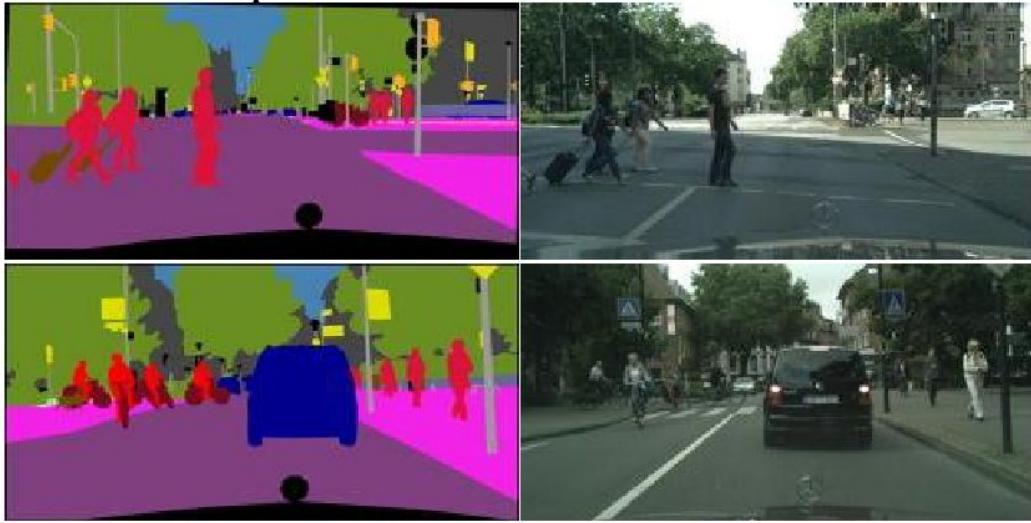
Pix2pix (PatchGAN)

Image-to-Image Translation with Conditional Adversarial Networks

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros

<https://arxiv.org/abs/1611.07004>

Motivations



Many tasks in computer vision and computer graphics require image-to-image ‘translation’: e.g. segmentation, pseudocoloring, texture filling.

Why not use a (more or less) conventional CNN?

CNNs learn to minimize a loss function – an objective that scores the quality of results – and although the learning process is automatic, a lot of manual effort still goes into designing effective losses. In other words, we still have to tell the CNN what we wish it to minimize. But, just like King Midas, we must be careful what we wish for!

If we follow a naive approach and ask the CNN to minimize the Euclidean distance between predicted and ground truth pixels,

1. it will tend to produce blurry results.
 - This is because Euclidean distance is minimized by averaging all plausible outputs, which causes blurring.
2. without z , the net could still learn a mapping from X to Y , but would produce deterministic outputs, and therefore fail to match any distribution other than a delta function (point distribution).
 - Conveys the essence of the generative approach: noise forces the learning process to form robust intermediate representations.

Why GANs? Why not $X \rightarrow Y$?

- Q: How to make the output indistinguishable from reality?
- A: Generative Adversarial Networks
 - But: How to make the generator generate the output that corresponds to the input image?

Some details in advance:

- Original GANs used Gaussian noise.
 - This did not prove effective.
- Pix2pix adds noise via dropout applied to G.
 - Also at test time!

Conditional GANs

- GANs learn a generative model of data
- Conditional GANs (cGANs) learn a conditional generative model.

GAN:

$$G : z \rightarrow y$$

where: y - prediction (image), z - random noise vector

Conditional GAN:

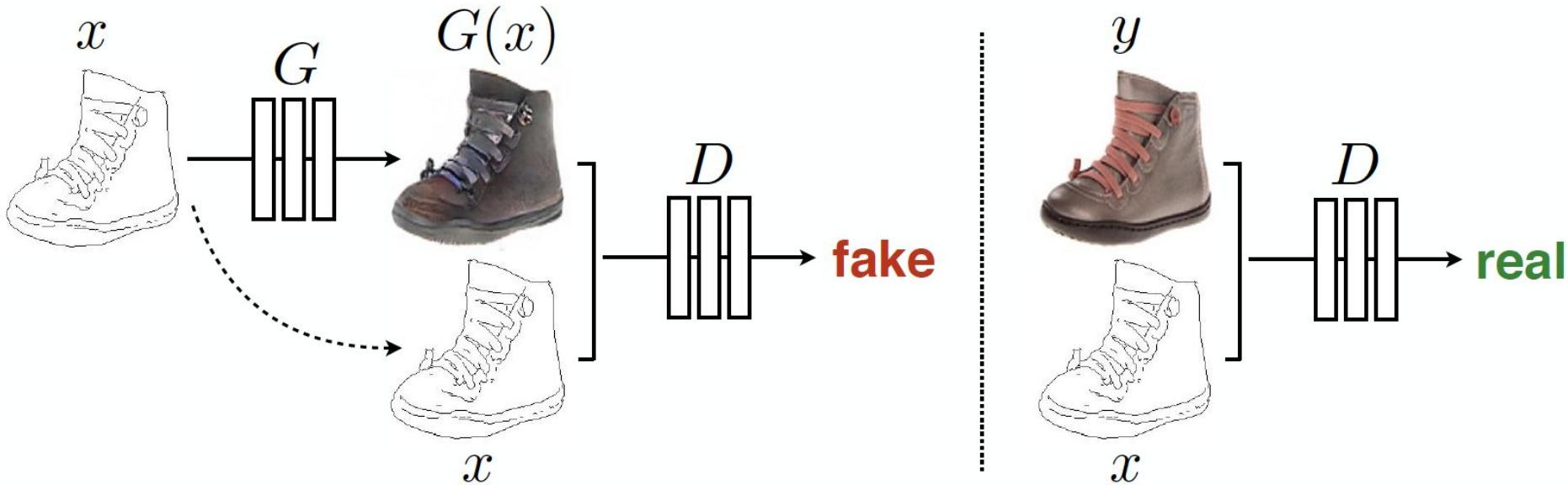
$$G : \{x, z\} \rightarrow y$$

where x is an additional input vector, conditioning variable (e.g., ‘simplified’ input image)

Discriminator: D: (x,y) -> {0,1} (fake vs. real)

- Notice: we feed also the conditioning vector x into D.

Training protocol



- Random component not shown in the diagram for clarity.
 - Both G and D have two inputs: $G(x,z)$, $D(x,y)$
- In practice, this setup assumes access to a ‘transformer’ from y to x
 - E.g. a Canny filter that finds contours in images.

Definition of the loss function

Loss function:

$$\begin{aligned}\mathcal{L}_{cGAN}(G, D) = & \mathbb{E}_{x,y}[\log D(x, y)] + \\ & \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z)))]\end{aligned}$$

In practice, combining with the distance from the true image helps:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x, z)\|_1]$$

(L2 caused more blurring)

The total loss:

$$G^* = \arg \min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$$

Control baseline

$$\begin{aligned}\mathcal{L}_{GAN}(G, D) = & \mathbb{E}_y[\log D(y)] + \\ & \mathbb{E}_{x,z}[\log(1 - D(G(x, z)))]\end{aligned}$$

Notice:

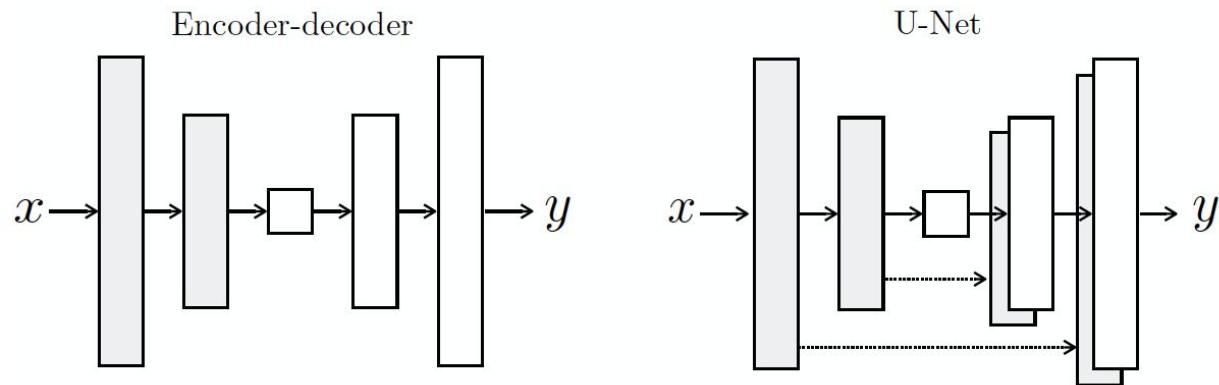
- The discriminator D is not conditioned on x (while G still is).
- As a result, D has no access to the information on the ‘category’ of object being generated by G.

The architecture

Basic building block for both G and D:

- convolution
- batch normalization
- ReLU

Two considered architectures for G (U-Net used in the experiments):



Motivation: desired spatial correspondence between x and y .

Discriminator

- Because L1 causes blurring, it already takes care of the low frequencies (overall ‘rough’ structure of the image).
- Therefore, the design of D focuses on detail (high frequencies).
- Thus, it’s enough for D to work with small image patches.
- D takes an NxN patch of the image and decides whether it comes from the real image (y) or fake ($G(x,z)$).
- Observations:
 - D is thus a ‘textural discriminator’, rather than a discriminator of entire images (and the associated loss can be considered texture loss).
 - D perceives the image as a Markov random field, with pixels more distant than patch’s diameter being considered as independent random variables.

Training

Training setup follows the original GAN paper and other earlier recommendations:

- One step of SGD on D interwoven with one step of SGD on G.
- D trained slower than G, by using $G^*/2$ loss
- Rather than minimize

$$\log(1 - D(x, G(x, z)))$$

maximize

$$\log D(x, G(x, z))$$

(notice that these two formulas are not equivalent)

Experimental domains

Unidirectional

- Architectural labels → photo
- Monochromatic photo → color photo
- Edges → photo
- Sketch → photo
- Day → night
- Thermal image → color photos
- Photo with missing pixels → inpainted photo

Bidirectional:

- Semantic labels ↔ photo
- Map ↔ aerial photo

Evaluation

Pixel-by-pixel distance (e.g., L1 or L2) too primitive to capture the quality of the produced output (and not penalize the variability resulting from the randomized nature of the generator).

Therefore, other evaluation methods:

1. Amazon Mechanical Turk (AMT): human subjects score the plausibility
 - a. The subject (*Turker*) is shown a pair of x and y and is expected to tell if it's real or fake.
2. Testing whether an off-shelf recognition system (e.g. AlexNet or GoogLeNet) can recognize objects in the synthesized images ('inception score')
 - a. Motivation: low probability score on all outputs of a model suggests that the input image y does not present a realistic scene/object.

Ablation experiments

Ablation: A type of experiment commonly conducted in deep learning:

1. The starting point/reference: the complete system.
2. Disable a component (*ablation*), e.g.
 - a. Eliminate a term in the loss function.
 - b. Disable an architectural component (e.g. batch normalization).
3. Compare the capabilities (e.g. values of metrics) of the ablated system with the complete one.

In this study (see next page):

- L1+cGAN: complete system
- L1: only the L1 loss used in training
- cGAN: only adversarial loss used in training

Ablation experiments



Ablation experiments

Loss	Per-pixel acc.	Per-class acc.	Class IOU
L1	0.42	0.15	0.11
GAN	0.22	0.05	0.01
cGAN	0.57	0.22	0.16
L1+GAN	0.64	0.20	0.15
L1+cGAN	0.66	0.23	0.17
Ground truth	0.80	0.26	0.21

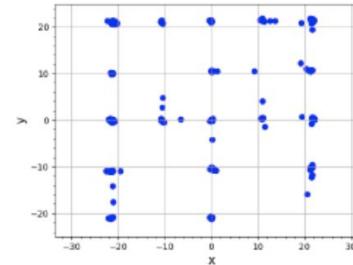
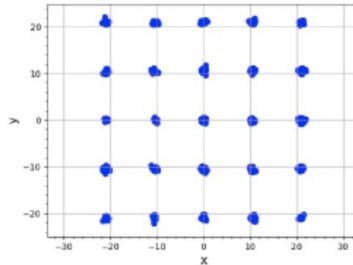
Table 1: FCN-scores for different losses, evaluated on Cityscapes labels↔photos.

Ordinary GAN performs very bad because it ‘collapses’ into producing almost the same output for all inputs (*mode collapse*).

Mode collapse

Mode collapse is a special case of a local Nash equilibrium characterized with suboptimal generative performance.

- Consider a multi-modal real 2D distribution below, left: 25 modes, hardly any samples in between.
 - Assume certain stage of generator training, with *some* modes already modeled (below, right)
 - The discriminator penalizes moving generated samples outside the modes, because such samples are easy to identify as fakes.
 - Therefore the generated samples remain where they are and don't migrate to other modes
- Therefore, GANs may find it difficult to capture global, multimodal sample density.

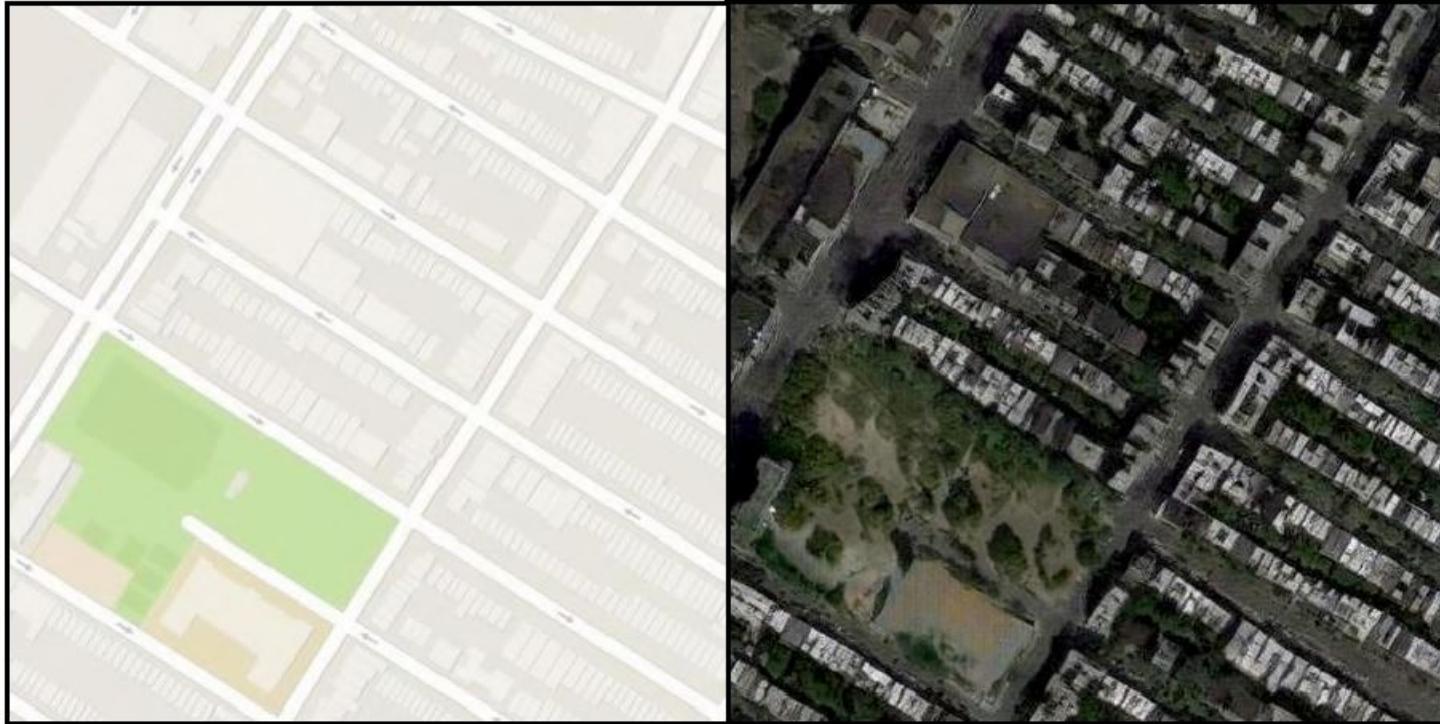


Observations

- Learns effectively even from small samples (e.g. 400 images in facade dataset)
 - In part because of the fact that training is patch-based: our samples are subwindows, not entire images.
- Computationally efficient (training time in the order of hours on Titan X GPU)
- Querying time under 1 second
- Many more experiments covered in the paper:
 - Impact of patch size: “From PixelGANs (1x1) to PatchGANs to ImageGANs (286x286)”
 - ‘Hallucinations’, i.e generating of structure in y in absence of evidence in x .
- Important practical aspect: By operating on subwindows, G is a fully convolutional network, so it can be applied to images of arbitrary dimensions.

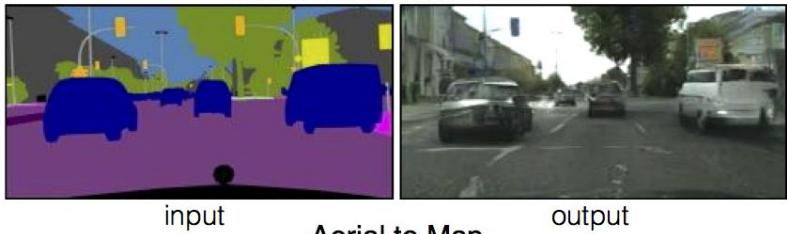
Exploiting the fully convolutional nature of G

G trained on 256x256, when tested on 512x512:



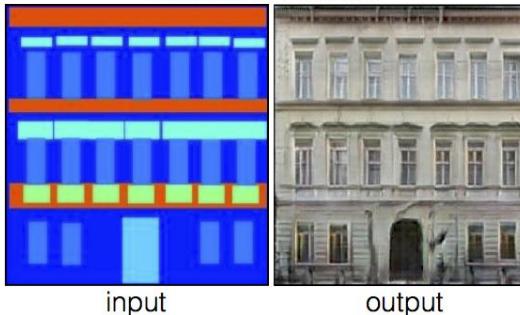
Other examples

Labels to Street Scene



Aerial to Map

Labels to Facade



BW to Color



Aerial to Map

The image displays two maps of a residential neighborhood side-by-side. The left map is a detailed satellite view, showing individual houses with distinct roofs, patches of green lawns, and a network of streets. The right map is a simplified version of the same area, where the buildings are represented by small, uniform white rectangles and the streets form a clear, intersecting grid of gray lines. This comparison illustrates how complex real-world data can be reduced into a more abstract or generalized representation.

input

output

Day to Night



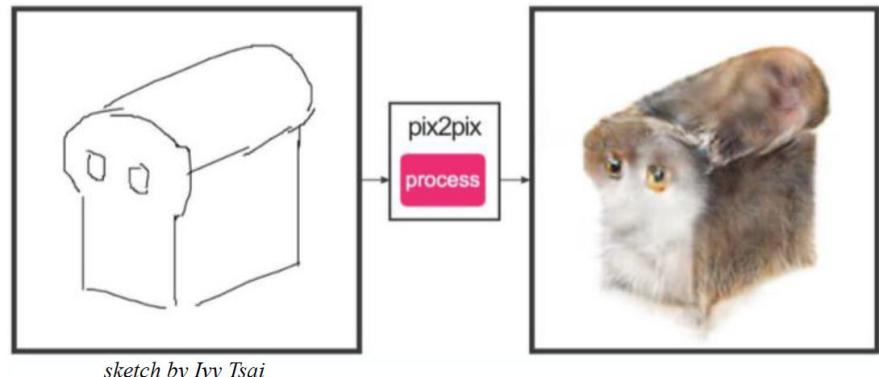
Edges to Photo



output

Other examples

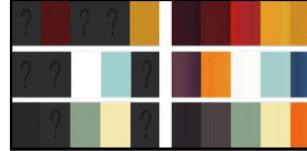
#edges2cats by Christopher Hesse



Background removal



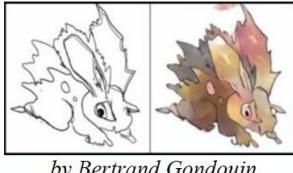
Palette generation



Sketch→Portrait



Sketch → Pokemon



"Do as I do"



#fotogenerator

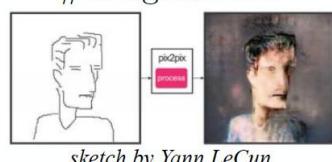
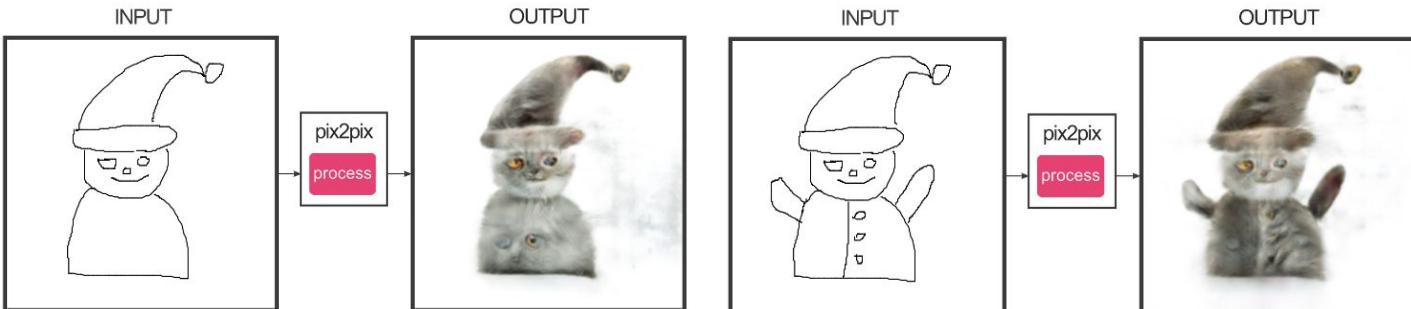


Figure 11: Example applications developed by online community based on our pix2pix codebase: #edges2cats [3] by Christopher Hesse, Background removal [6] by Kaihu Chen, Palette generation [5] by Jack Qiao, Sketch → Portrait [7] by Mario Klingemann, Sketch→Pokemon [1] by Bertrand Gondouin, “Do As I Do” pose transfer [2] by Brannon Dorsey, and #fotogenerator by Bosman et al. [4].



Human-drawn sketches to color images



Thermal images to RGB

Input



Ground truth



Output



Input



Ground truth



Output





Figure 12: *Learning to see: Gloomy Sunday*: An interactive artistic demo developed by Memo Akten [8] based on our pix2pix codebase. Please click the image to play the video in a browser.

<https://vimeo.com/260612034>

CycleGAN

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Jun-Yan Zhu, Taesung Park, Phillip Isola, Alexei A. Efros

<http://arxiv.org/abs/1703.10593>

The task of image-to-image translation

Train a mapping:

$$G : X \rightarrow Y$$

such that the output

$$\hat{y} = G(x)$$

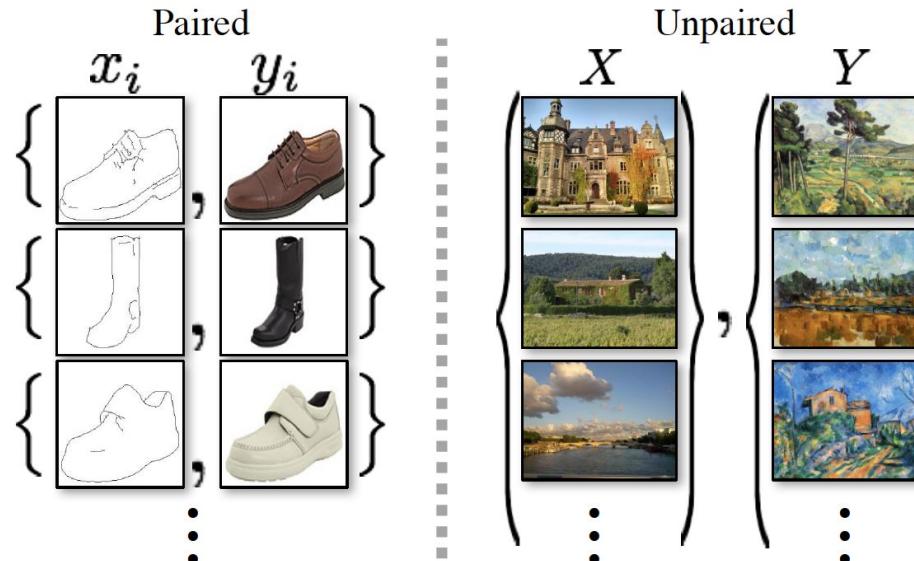
is indistinguishable from images $y \in Y$ by an adversary D trained to tell apart \hat{y} from y .

Note:

- purely adversarial setting (random input component z temporarily dropped from our line of argument),
- in general applicable beyond images.

Motivation

- Paired examples (x, y) for image-to-image translation tasks (e.g. approached with cGAN) are hard to come by.
 - And/or they can be biased by some automated way in which x s are obtained from y s (e.g. Canny edge detector to obtain sketches from photos).
- Can we learn translation *between entire domains/sets, without pairing of examples?*



The challenges

Q: Could we realize this task by, e.g., simply training a cGAN model (conditional setting) on randomly paired examples drawn from the domains?

A: No, because:

1. No guarantee that an individual input x and output y are paired up in a meaningful way.
 - o There are infinitely many mappings G that will induce the same distribution in Y .
 - o In other words: a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution.
2. In practice, it is difficult to optimize the adversarial objective in isolation:
 - o Problem of *mode collapse*: all input images are mapped to the same output image; optimization fails to make progress.

Solution: Cycle consistency

Conceptual motivation: In an ideal scenario, translating a sentence from English to French, and then back from French to English, should result in the original sentence.

Therefore: train two translators (conditional generators) simultaneously:

$$G : X \rightarrow Y$$

$$F : Y \rightarrow X$$

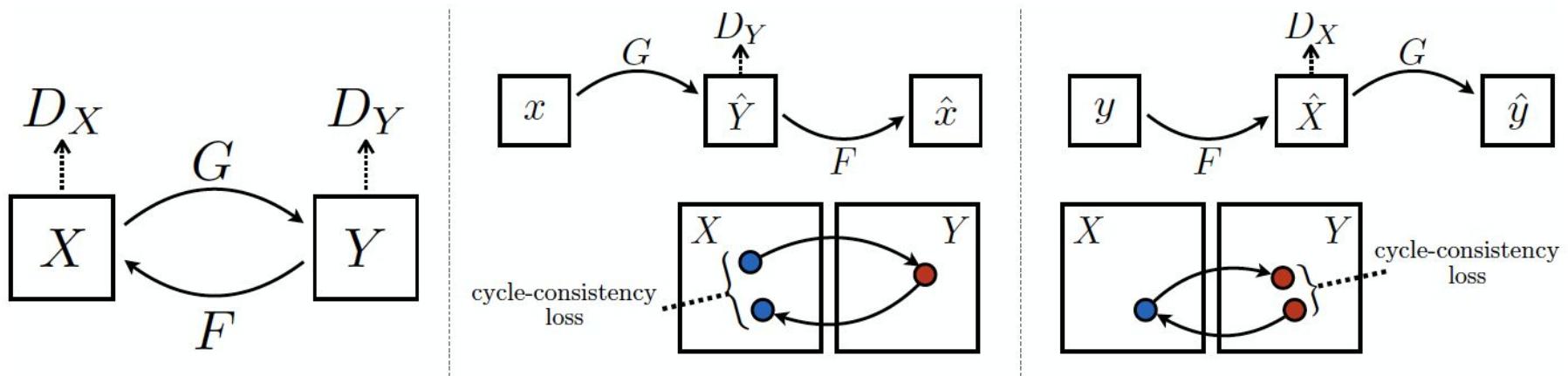
and apply simultaneously *cycle consistency loss* that encourages:

$$x \rightarrow G(x) \rightarrow F(G(x)) \approx x$$

$$y \rightarrow F(y) \rightarrow G(F(y)) \approx y$$

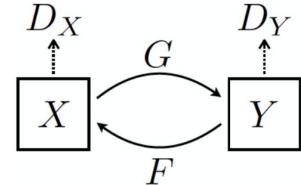
Cycle consistency (CycleGAN): The complete model

Cycle consistency tunes the generators so that they become close to being their mutual inverses ($F \approx G^{-1}$).



Loss definition

Adversarial component for G (and analogously for F):



$$\begin{aligned}\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) &= \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] \\ &\quad + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log(1 - D_Y(G(x)))]\end{aligned}$$

Cycle consistency loss:

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G, F) &= \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] \\ &\quad + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1].\end{aligned}$$

Total loss (the authors used $\lambda=10$):

$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) &= \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &\quad + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &\quad + \lambda \mathcal{L}_{\text{cyc}}(G, F),\end{aligned}$$

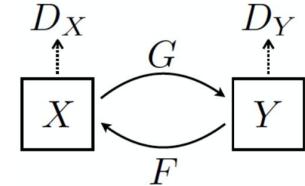
The task: solve

$$G^*, F^* = \arg \min_{G, F} \max_{D_x, D_Y} \mathcal{L}(G, F, D_X, D_Y)$$

Comments

Advantages:

- No pairing required.
- No need for a task-specific similarity function between input and output.
- The input and the output (x,y) do not have to belong in the same space.



Notice:

- This can be viewed as training two autoencoders in parallel:

$$F \circ G : X \rightarrow X$$

$$G \circ F : Y \rightarrow Y$$

- However, the intermediate representations are ‘enforced’ here.
- Also related to: adversarial autoencoders.

Architecture

Generators:

- stride-2 convolutions,
- residual blocks
- fractionally-strided convolutions
- 6 blocks for 128x128 images, 9 blocks for 256x256 images.

Discriminators:

- Patch-level
- 70 x 70 PatchGANs (i.e., Pix2pix, Image-to-Image Translation with Conditional Adversarial Networks, Isola et al.)

Training

1. Rather than using the original log-likelihood, uses least-square loss:

- More stable in training.
- Produces better results.

Technically: training G to minimize

$$\mathbb{E}_{x \sim p_{\text{data}}(x)} [(D(G(x)) - 1)^2]$$

and D to minimize

$$\mathbb{E}_{y \sim p_{\text{data}}(y)} [(D(y) - 1)^2] + \mathbb{E}_{x \sim p_{\text{data}}(x)} [D(G(x))^2]$$

2. Adam algorithm, batch size = 1 (! this is really puzzling; the authors probably had to modify Adam)

Training

3. Addressing “model oscillation” (G and D moving back and forth between states in the search space):

- Updating discriminators using a history of generated images, rather than only those produced by the latest generators.

Expressed in game-theoretical terms:

- Successful training of a standard GAN produces a generator that is well-tuned to win against ('fool') a *specific discriminator*.
- Ideally, we would like to obtain a generator that wins against *all discriminators*.
- The above training on (additionally) fakes generated by the past ('historic') generators approximates takes us closer to that goal.
- This is closely related to the concept of solution concept in game theory.
 - Defines the properties of a ‘successful player’.
 - Popular solution concepts: Nash equilibrium, Maximization of Expected Utility.

Input x

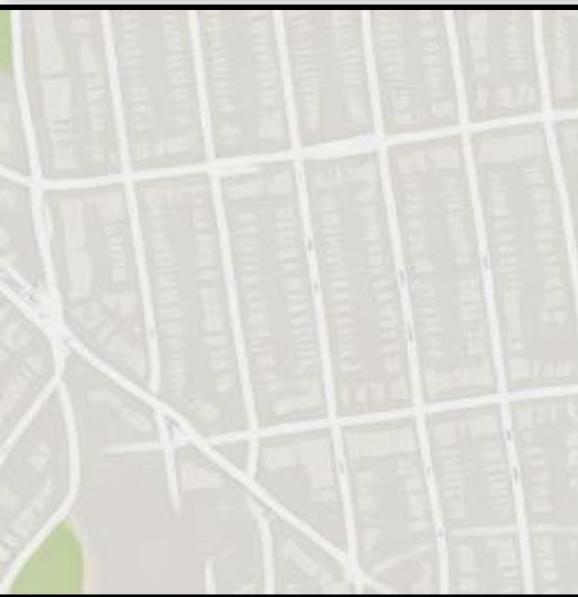
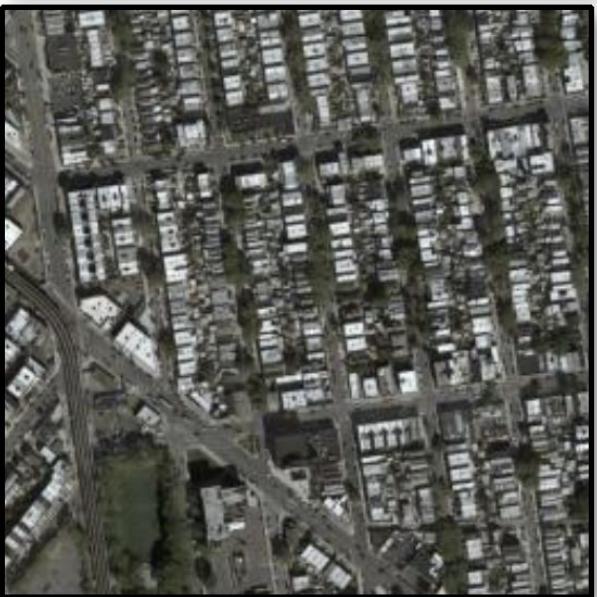


Output $G(x)$



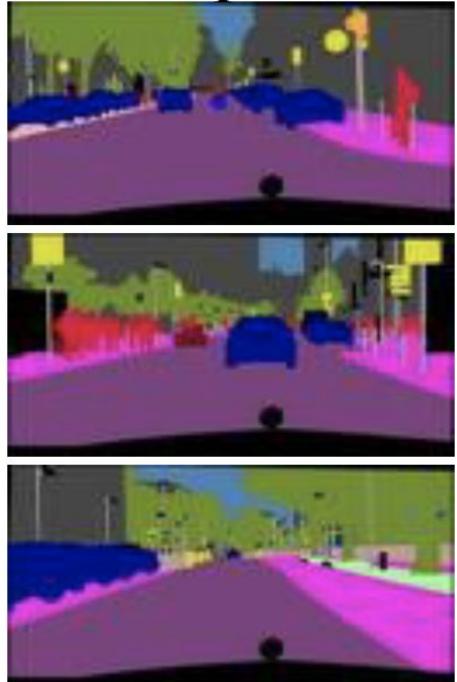
Reconstruction $F(G(x))$





Results

Input



CycleGAN



pix2pix



Ground truth



Notice: pix2pix uses paired examples!

Metrics

- Human assessment: Amazon Mechanical Turk (AMT) perceptual study:
 - Subject shown pairs of images (real, fake) and asked to click the image they thought was real.
- FCN score:
 - Applies an FCN network (ConvNet) to perform semantic segmentation of the scene.
 - Comparing the returned labels to ground truth labels (IoU?)

Loss	Map → Photo	Photo → Map
	% Turkers labeled <i>real</i>	% Turkers labeled <i>real</i>
CoGAN [32]	0.6% ± 0.5%	0.9% ± 0.5%
BiGAN/ALI [9, 7]	2.1% ± 1.0%	1.9% ± 0.9%
SimGAN [46]	0.7% ± 0.5%	2.6% ± 1.1%
Feature loss + GAN	1.2% ± 0.6%	0.3% ± 0.2%
CycleGAN (ours)	26.8% ± 2.8%	23.2% ± 3.4%

Table 1: AMT “real vs fake” test on maps↔aerial photos at 256×256 resolution.

Loss	Per-pixel acc.	Per-class acc.	Class IOU
CoGAN [32]	0.40	0.10	0.06
BiGAN/ALI [9, 7]	0.19	0.06	0.02
SimGAN [46]	0.20	0.10	0.04
Feature loss + GAN	0.06	0.04	0.01
CycleGAN (ours)	0.52	0.17	0.11
pix2pix [22]	0.71	0.25	0.18

Table 2: FCN-scores for different methods, evaluated on Cityscapes labels→photo.

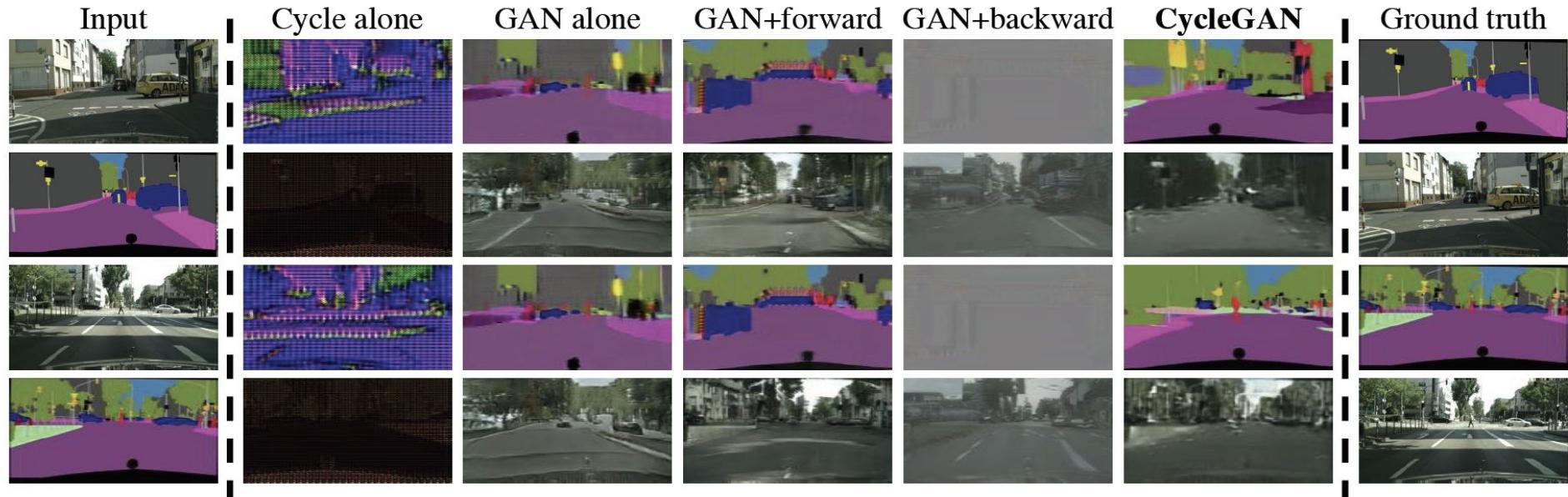


Figure 7: Different variants of our method for mapping labels \leftrightarrow photos trained on cityscapes. From left to right: input, cycle-consistency loss alone, adversarial loss alone, GAN + forward cycle-consistency loss ($F(G(x)) \approx x$), GAN + backward cycle-consistency loss ($G(F(y)) \approx y$), CycleGAN (our full method), and ground truth. Both *Cycle alone* and *GAN + backward* fail to produce images similar to the target domain. *GAN alone* and *GAN + forward* suffer from mode collapse, producing identical label maps regardless of the input photo.

Input



Monet



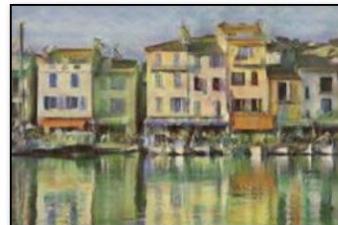
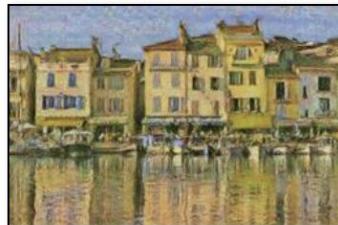
Van Gogh



Cezanne



Ukiyo-e



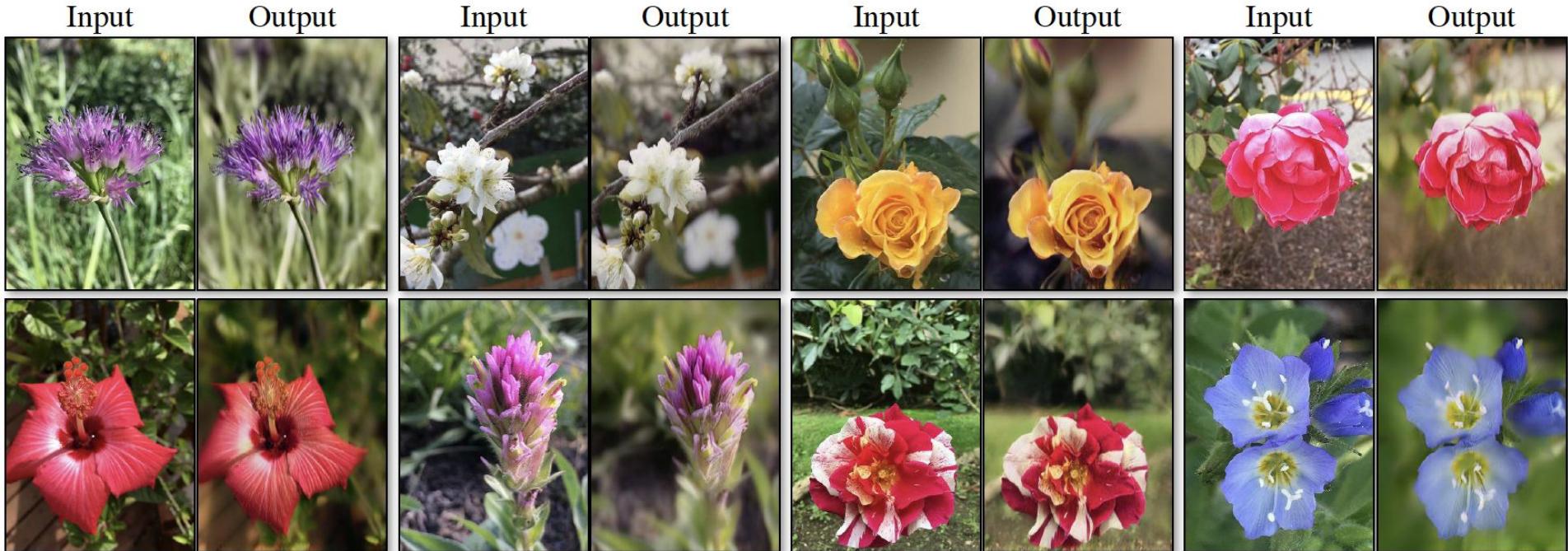


Figure 14: Photo enhancement: mapping from a set of smartphone snaps to professional DSLR photographs, the system often learns to produce shallow focus. Here we show some of the most successful results in our test set – average performance is considerably worse. Please see our [website](#) for more comprehensive and random examples.

Typical failures

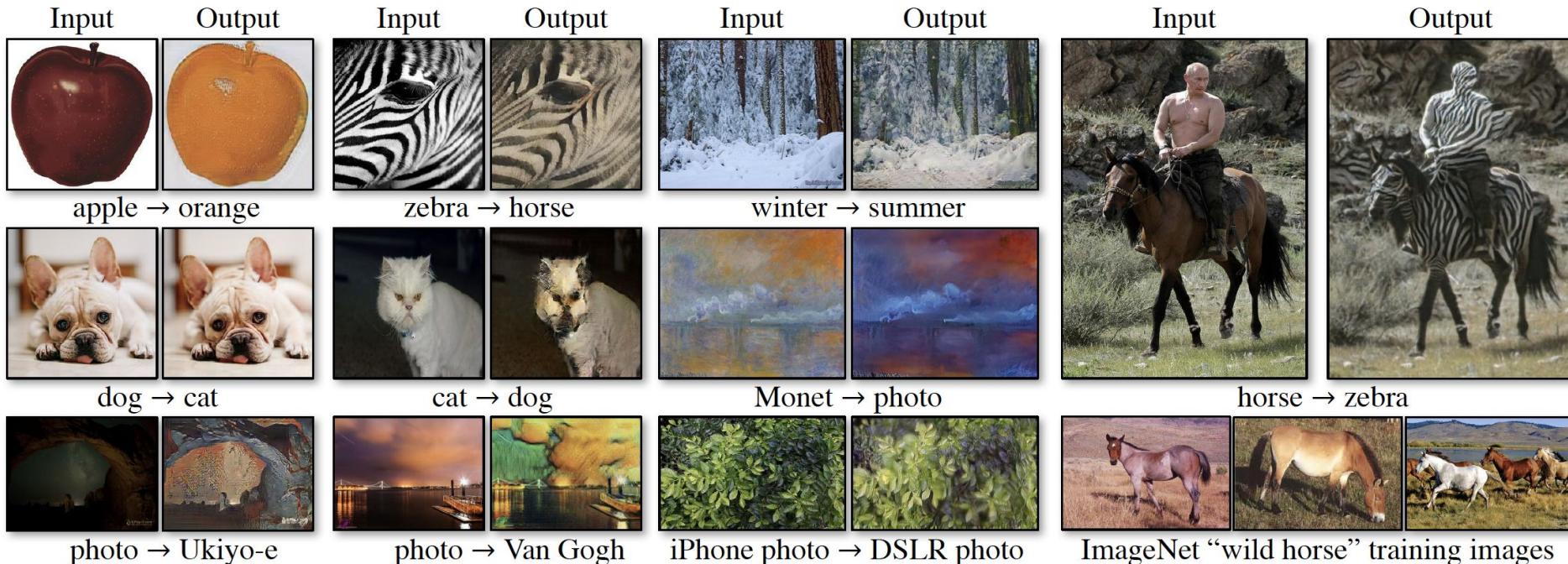


Figure 17: Typical failure cases of our method. Left: in the task of dog→cat transfiguration, CycleGAN can only make minimal changes to the input. Right: CycleGAN also fails in this horse → zebra example as our model has not seen images of horseback riding during training. Please see our [website](#) for more comprehensive results.

Comments

- More baseline algorithms used in the paper (CoGAN, SimGAN, BiGAN/ALI)

Observation (more general):

- Notice how different the 'fake-real' discrimination task is from the typical classification tasks.
- The discriminator must operate on a different level of abstraction, because the probability distribution to be learned here typically comprises a much broader set of concepts (e.g. *landscape photos*) than decision classes used in typical classification tasks (e.g. *tree*).

Deep Convolutional GAN (DCGAN)

Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks

Alec Radford, Luke Metz, Soumith Chintala

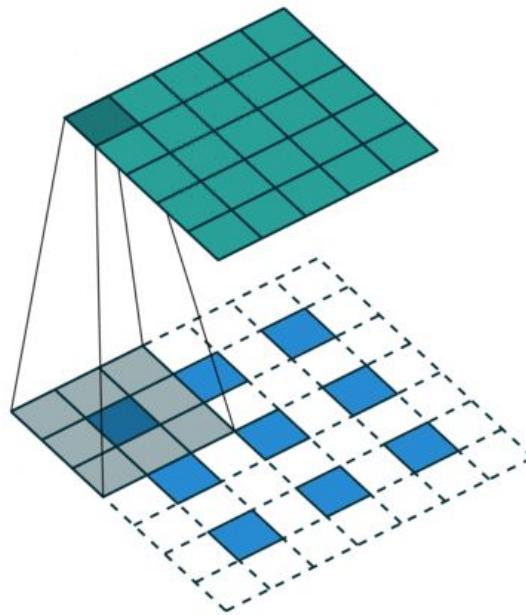
<https://arxiv.org/abs/1511.06434>

Motivations

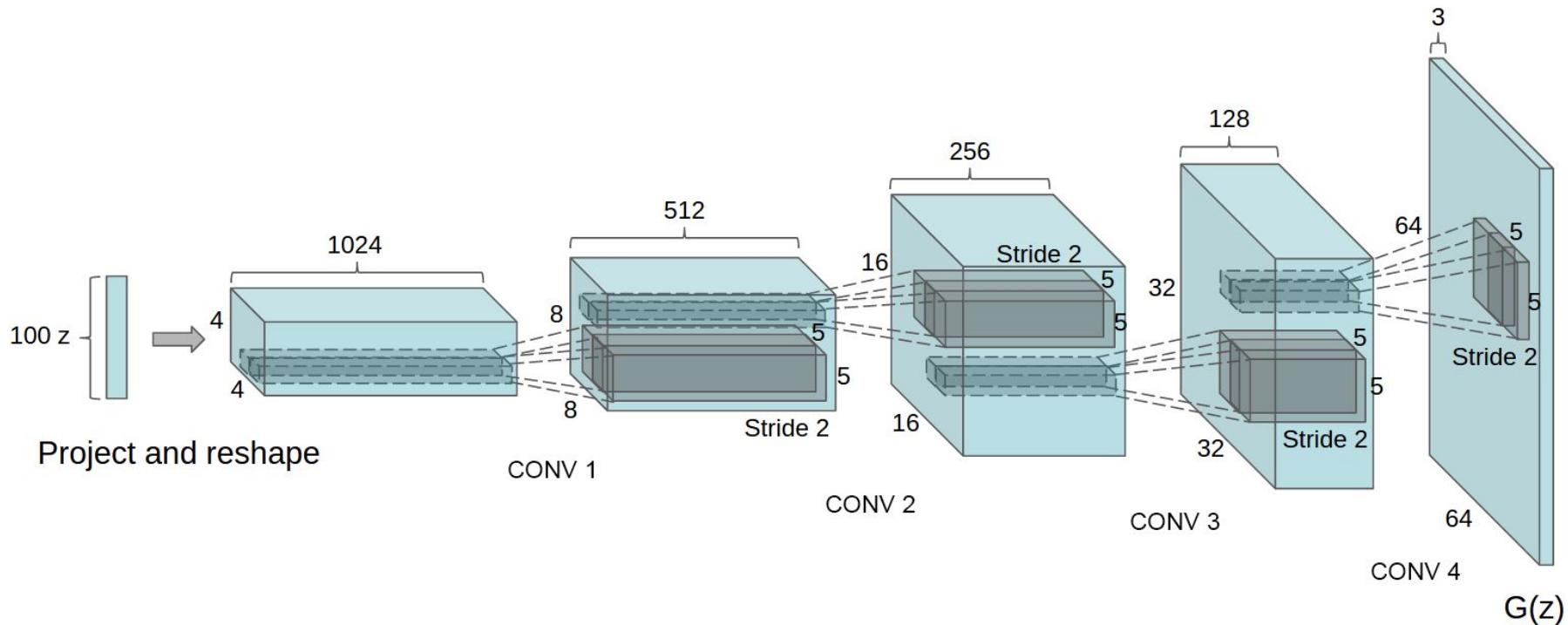
Come up with guidelines for designing GANs that are stable to train: “constraints on the architectural topology of Convolutional GANs”

- Replace pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
 - Allows the network to learn its own spatial downsampling.
- Use batch normalization in both the generator and the discriminator.
 - Exception: Applying batchnorm to all layers resulted in sample oscillation and model instability.
 - Therefore: no batchnorm in generator’s output layer and discriminator’s input layer.
- Remove fully connected hidden layers for deeper architectures.
 - Except for the first layer of G , which multiplies z by a matrix and reshapes it for convolution.
- Use ReLU activation in generator for all layers
 - Except for the output, which uses Tanh: bounded activation allowed the to faster saturate and cover the color space of the training distribution.
- Use LeakyReLU activation in the discriminator for all layers.

Fractionally strided convolution



Generator architecture



Results

Massive experimental investigation; among others:

- Generating bedrooms: Large-scale Scene Understanding (LSUN) dataset
- Classifying CIFAR-10 and numbers in StreetView House Numbers dataset (SVHN) using GANs as a feature extraction.
- Human faces (scrapped from the internet, 10k people, 3M images)
- Visualization of features

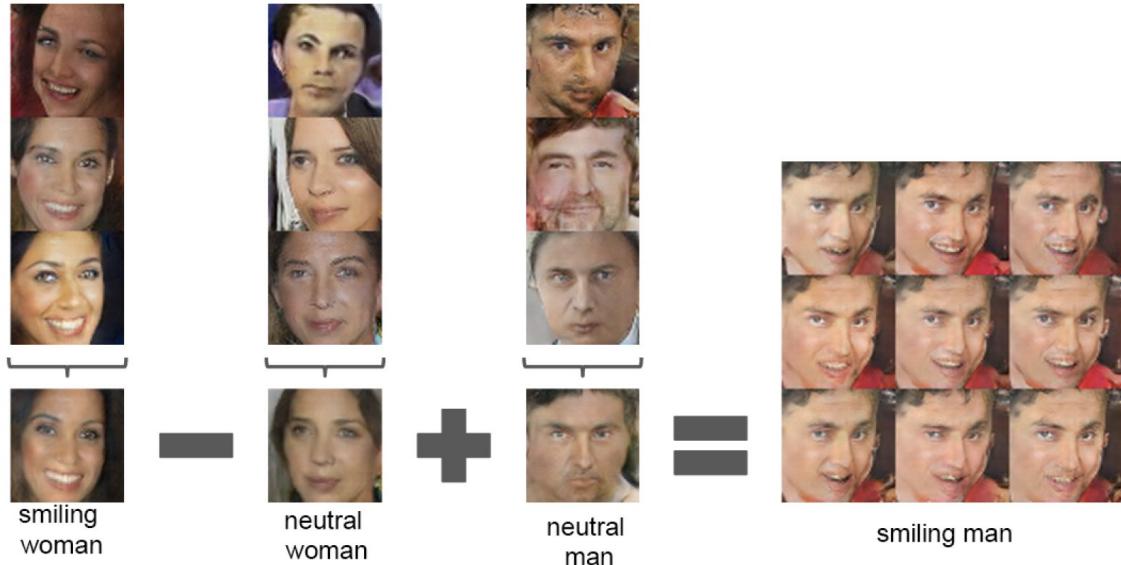
Digression: Interesting preprocessing for the bedrooms problem: autoencoder for dataset deduplication.

- Trained de-noising (?) dropout regularized ReLU autoencoder on 32x32 downsampled center-crops of training examples.
- Latent ('code') layer binarized via thresholding, obtaining so a form of semantic hash code.
- Semantic hash codes then used to identify very similar images.
- FP rate < 0.01. Helped removing nearly 275,000 [near] duplicates.

Results: Some interpretability of the latent

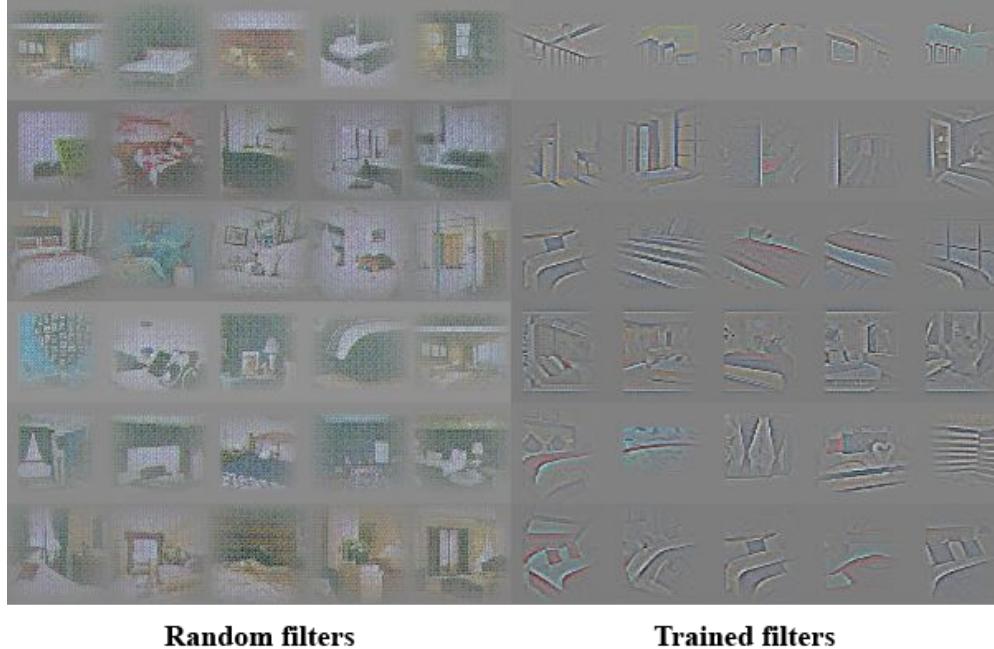
All images shown in this slide are fakes generated by G.

1. Z vectors corresponding to the images shown are averaged over columns.
2. Arithmetic performed on the mean vectors, creating a new vector Z' .
3. The center sample on the right produced by querying G on Z' .
4. Uniform noise $+/-0.25$ added to Z' to produce the 8 other samples (to show interpolation capabilities).



The resulting sample and 8 similar samples obtained by disturbing Z' with a random noise and feeding it into G

Visualization of features



- Right: guided backpropagation (Springenberg et al., 2014) visualizations of maximal axis-aligned responses for the first 6 learned convolutional features from the last convolution layer in the discriminator.
- Notice how localized the features are (recall: last convolutional layer!)

Coulomb GANs

Coulomb GANs: Provably Optimal Nash Equilibria via Potential Fields

Thomas Unterthiner, Bernhard Nessler, Calvin Seward, Günter Klambauer, Martin Heusel, Hubert Ramsauer, Sepp Hochreiter

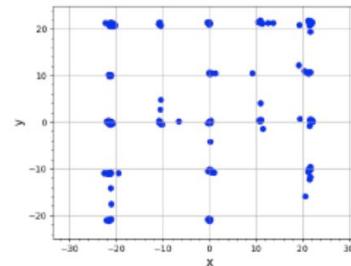
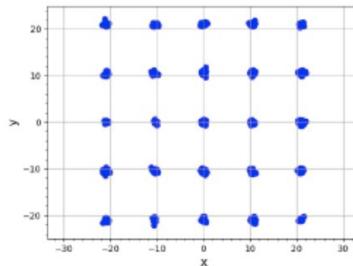
<http://arxiv.org/abs/1708.08819>

(a representative of many other variants of GANs; not necessarily particularly popular today)

Motivation: Mode collapse (reminder)

Mode collapse is a special case of a local Nash equilibrium associated with suboptimal generative performance.

- Consider a multi-modal real 2D distribution (below, left: 25 modes, hardly any samples in between)
 - Assume certain stage of generator training, with *some* modes already modeled (below, right)
 - The discriminator penalizes moving generated samples outside the modes, because such samples are easy to identify as fakes.
 - Therefore the generated samples remain where they are and don't migrate to other modes
- Therefore, GANs may find it difficult to capture global, multimodal sample density.



Motivation

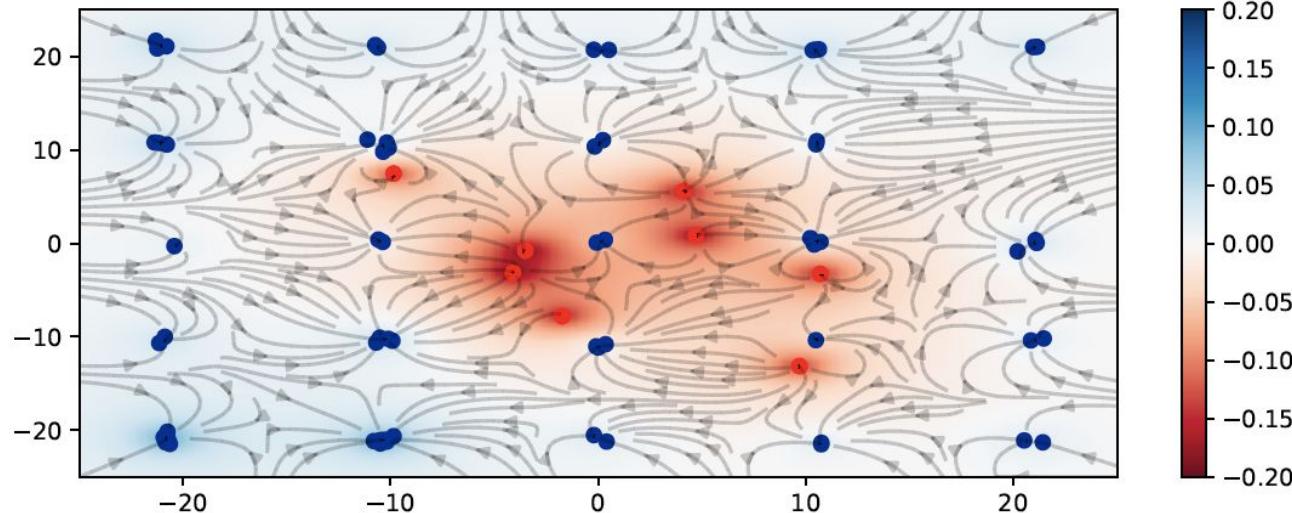
- It has been proven that GAN learning does converge when discriminator and generator are learned using a two time-scale learning rule (Heusel et al., 2017).
 - Convergence means that the expected SGD-gradient of both the discriminator objective and the generator objective are zero.
 - Thus, neither the generator nor the discriminator can locally improve, i.e., learning has reached a local Nash equilibrium.
- However, convergence alone does not guarantee good generative performance.
 - It is possible to converge to sub-optimal solutions which are local Nash equilibria.

Authors' proposal:

- Coulomb GAN, which has only one Nash equilibrium.
- Show that this Nash equilibrium is optimal, i.e., the model distribution matches the target distribution.

Coulomb GAN: The intuition

True samples (blue) and generated samples (red) create a potential field (scalar field). True samples act as sinks that attract the generated samples, which repel each other. The superimposed vector field shows the forces acting on the generator samples to equalize potential differences, and the background color shows the potential at each position.



Realization

Rather than minimizing the difference* between densities $\rho(a) = p_y(a) - p_x(a)$, where $p_x(a)$ is the model density and $p_y(a)$ is the target density, minimize a potential function defined via kernel functions $k(a,b)$ that define the influence of a point at b onto a point at a :

$$\Phi(\mathbf{a}) = \int \rho(\mathbf{b}) k(\mathbf{a}, \mathbf{b}) d\mathbf{b}, \quad k(\mathbf{a}, \mathbf{b}) = \frac{1}{(\sqrt{\|\mathbf{a} - \mathbf{b}\|^2 + \epsilon^2})^d}$$

Minimizing this potential function causes gradual shifting of the generated samples and, consequently, minimization of the difference of densities $\rho(a)$.

* In conventional GANs, $D(a)$ is optimized to approximate the probability of seeing a target sample, or $\rho(a)$.

Technical realization

For N_x fakes (transformed by the generator) and N_y real samples, an unbiased estimate of Φ is:

$$\hat{\Phi}(\mathbf{a}) = \frac{1}{N_y} \sum_{i=1}^{N_y} k(\mathbf{a}, \mathbf{y}_i) - \frac{1}{N_x} \sum_{i=1}^{N_x} k(\mathbf{a}, \mathbf{x}_i)$$

Update rule for the discriminator:

$$d\mathbf{w} \leftarrow \nabla_{\mathbf{w}} \left[\frac{1}{2} \sum_{i=1}^{N_x} (D(\mathbf{x}_i) - \hat{\Phi}(\mathbf{x}_i))^2 + \frac{1}{2} \sum_{i=1}^{N_y} (D(\mathbf{y}_i) - \hat{\Phi}(\mathbf{y}_i))^2 \right]$$

Update rule for the generator (note: G is implicit in this formula):

$$d\boldsymbol{\theta} \leftarrow \nabla_{\boldsymbol{\theta}} \left[-\frac{1}{2} \frac{1}{N_x} \sum_{i=1}^{N_x} D(\mathbf{x}_i) \right]$$

(more technical details in the paper).

Results

- Tested on images CelebA, LSUN bedrooms, and CIFAR-10.
- Coulomb GANs tend to outperform standard GAN approaches like BEGAN and DCGAN, but are outperformed by the Improved Wasserstein GAN.
 - However Improved Wasserstein GAN used a more advanced network architecture based on ResNet blocks (Gulrajani et al., 2017).
- Results in terms of FID, Frechet Inception Distance (Frechet = Wasserstein-2):
 - Like inception score, but takes into account the distributions.

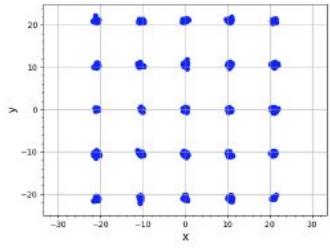
data set	BEGAN	DCGAN	WGAN-GP	MMD	ours
CelebA	29.2 / 28.5	21.4 / 12.5	4.8 / 4.2	63.2	9.3
LSUN	113 / 112	70.4 / 57.5	20.5 / 9.5	94.9	31.2
CIFAR10	-	-	29.3 / 24.8	38.2	27.3

Fréchet Inception Distance

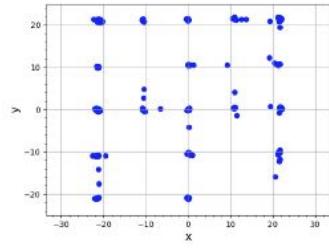
- Frechet Inception Distance (Frechet = Wasserstein-2):
 - Like inception score, but takes into account the distributions.
- Intuition behind the Frechet Distance:
 - *Imagine a man traversing a finite curved path while walking his dog on a leash, with the dog traversing a separate one. Each can vary their speed to keep slack in the leash, but neither can move backwards. The Fréchet distance between the two curves is the length of the shortest leash sufficient for both to traverse their separate paths. Note that the definition is symmetric with respect to the two curves—the Frechet distance would be the same if the dog was walking its owner. [Wikipedia]*

See also:

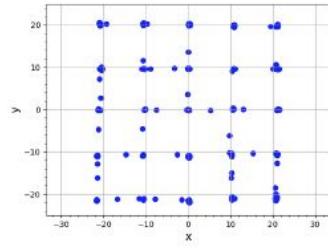
https://en.wikipedia.org/wiki/Fr%C3%A9chet_inception_distance



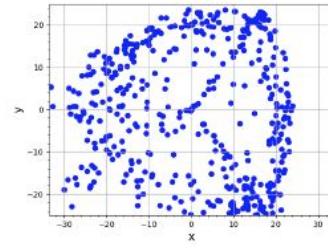
(a) True data



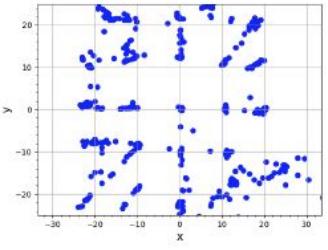
(b) GAN



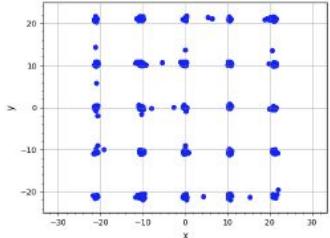
(c) Geometric GAN



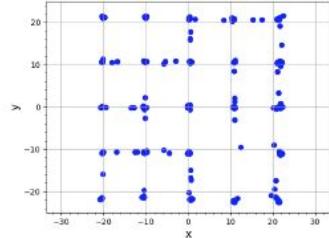
(d) WGAN



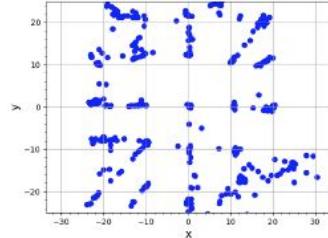
(e) meanGAN + proj.



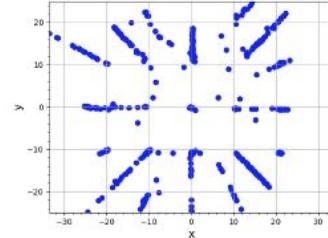
(f) Coulomb GAN



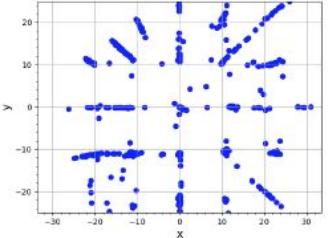
(g) GAN + WD



(h) Geo. GAN + WD



(i) WGAN + WD



(j) meanGAN + WD

Wasserstein GAN

Wasserstein GAN

Martin Arjovsky, Soumith Chintala, Léon Bottou

<https://arxiv.org/abs/1701.07875>

Motivation

Learning a generator in GAN is learning a probability distribution using a parameterized family of densities P_θ (m examples):

$$\max_{\theta \in \mathbb{R}^d} \frac{1}{m} \sum_{i=1}^m \log P_\theta(x^{(i)})$$

The authors:

- Notice that training a traditional GAN aims at minimizing the KL divergence between P_θ and the real distribution P_r (which realizes the above minimization).
- Discuss the pros and cons of various measure of the similarity of P_θ to P_r .
- Provide a comprehensive theoretical analysis of how the Earth Mover's Distance (EMD) behaves in comparison to popular probability distances and divergences used in the context of learning distributions.
- Propose Wasserstein GAN that minimizes a reasonable and efficient approximation of the EMD.

Earth Mover's Distance (EMD)

Informal definition:

- Given two distributions, one can be seen as a mass of earth properly spread in space, the other as a collection of holes in that same space.
- EMD measures the least amount of work needed to fill the holes with earth.
- A unit of work corresponds to transporting a unit of earth by a unit of ground distance.

For discrete random variables, can be computed via solving the corresponding transportation problem.

Adopted from:

https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/RUBNER/emd.htm

Distances/divergence studied in the paper

Total Variation (TV) distance (Σ - family of Borel subsets of the space [of images]):

$$\delta(\mathbb{P}_r, \mathbb{P}_g) = \sup_{A \in \Sigma} |\mathbb{P}_r(A) - \mathbb{P}_g(A)|$$

Kullback-Leibler (KL) divergence:

$$KL(\mathbb{P}_r \| \mathbb{P}_g) = \int \log \left(\frac{P_r(x)}{P_g(x)} \right) P_r(x) d\mu(x)$$

Jensen-Shannon (JS) divergence:

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r \| \mathbb{P}_m) + KL(\mathbb{P}_g \| \mathbb{P}_m)$$

Earth-Mover (EM) distance (Wasserstein-1):

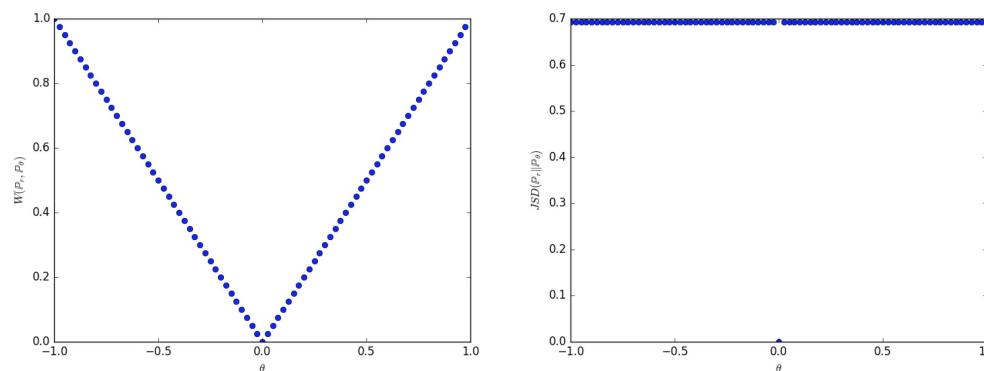
$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

Distance/divergence characteristics

Assume:

- $Z \sim U[0, 1]$: the uniform distribution on the unit interval.
- P_0 : the distribution of $(0, Z) \in \mathbb{R}^2$ (a 0 on the x-axis and the random variable Z on the y-axis), uniform on a straight vertical line passing through the origin.
- $g_\theta(z) = (\theta, z)$ with θ a single real parameter.

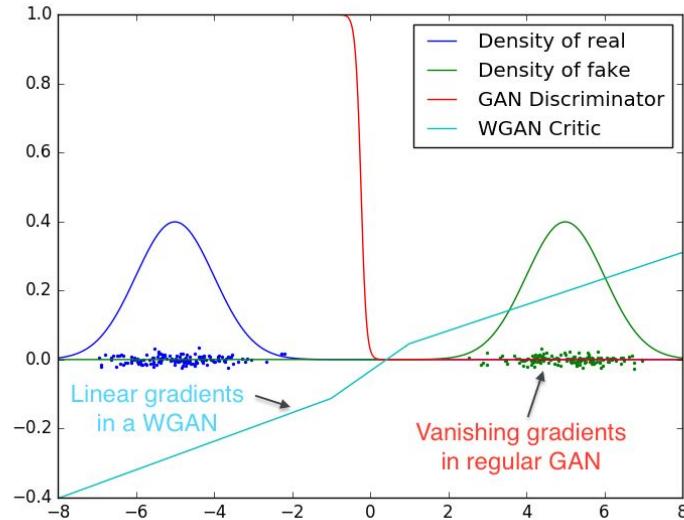
Plots of distances a function of distribution parameter θ , for Wasserstein and JSD:



KLD is zero everywhere, except for $\theta=0$, where its infinite.

Behavior at discrimination

Optimal discriminator and critic when learning to differentiate two Gaussians



WGAN's loss function maintains gradients 'everywhere' and so facilitates learning.

Wasserstein GAN

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size.

n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

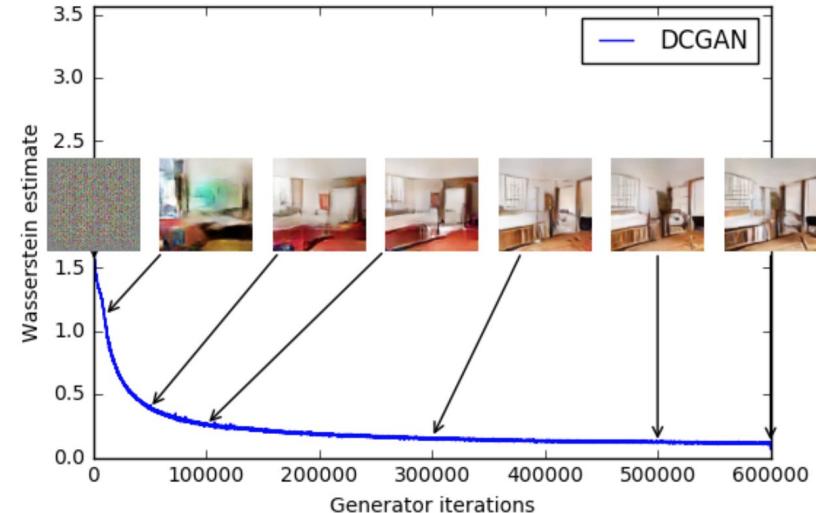
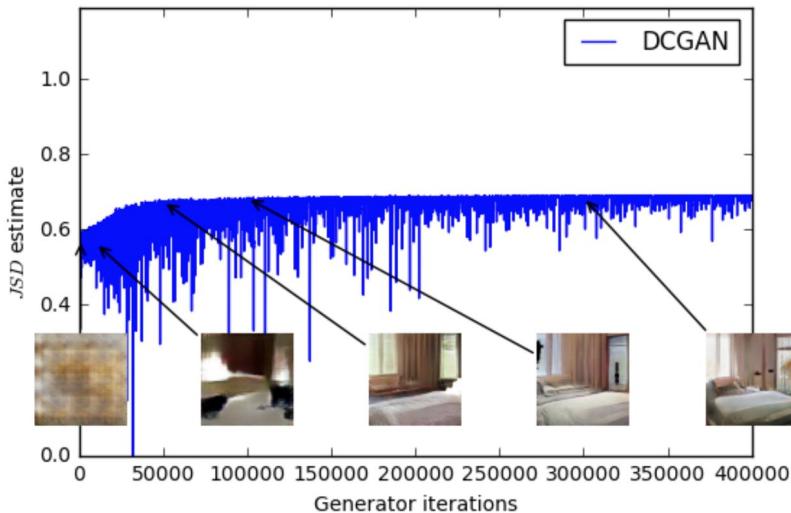
```
1: while  $\theta$  has not converged do
2:   for  $t = 0, \dots, n_{\text{critic}}$  do
3:     Sample  $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$  a batch from the real data.
4:     Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
5:      $g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$ 
6:      $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$ 
7:      $w \leftarrow \text{clip}(w, -c, c)$ 
8:   end for
9:   Sample  $\{z^{(i)}\}_{i=1}^m \sim p(z)$  a batch of prior samples.
10:   $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$ 
11:   $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$ 
12: end while
```

- Note: no logarithms; ‘raw’ discriminator output.
- Key element: weight clipping, to approximate Lipschitz continuity.

Implications

- a meaningful loss metric that correlates with the generator's convergence and sample quality
- improved stability of the optimization process

Shown empirically: left: JSD, right: Wasserstein.



General comments

- Wasserstein distance ‘wins’ because it always works, no matter how distant are the modes (‘masses’) in probability distributions.
- Lesson learned: loss functions matter.
 - It’s always worth to ask oneself: how will my loss function behave in extreme/degenerate cases?
- However: practical usefulness varies.
 - Mixed experience of the author of these slides.
 - Possible explanation: while EMD can provide better guidance for the optimization/learning algorithm, the *dynamics* of the two-person game between the generator and discriminator remains very complex.

Module 3: Autoencoders

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2019-2024

<http://www.cs.put.poznan.pl/kkrawiec/>



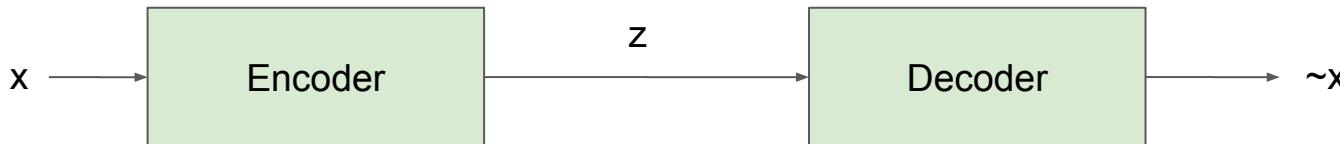
Outline

1. [Introduction](#)
2. [Autoencoders \(regular\)](#)
3. [Variational autoencoders](#)
4. Other autoencoders

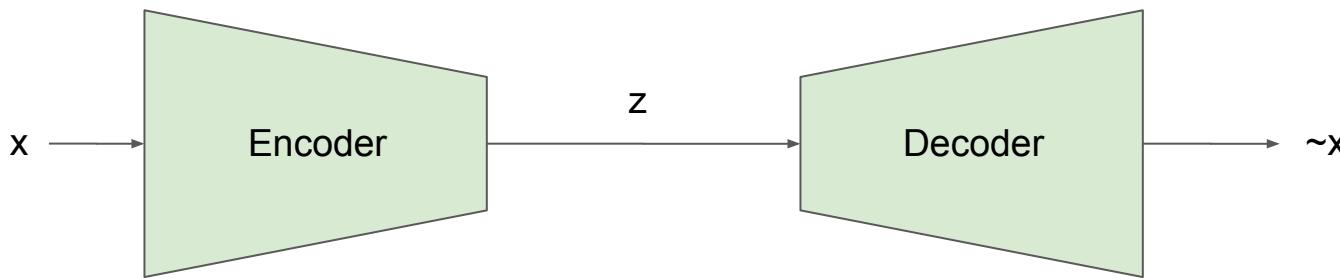
Introduction

General architecture of autoencoders (AE)

Composition of encoder $f:X \rightarrow Z$ and decoder $g:Z \rightarrow X$, where $Z \neq X$.



Or more illustratively:



The dimensionality of the latent space Z is typically much lower than the dimensionality of the input space X (often by orders of magnitude).

Introduction

- The name ‘autoencoder’ is meant to capture two aspects:
 - autoassociation,
 - encoding (compression of the input space X to the latent space Z).
- Training mode: Autoassociation:
 - The goal: acquire the capability to build associations between examples in X, i.e. (x, x')
 - In other words: approximate the identity function, but in a nontrivial way.
- Autoassociation does not require ‘encoding’ in general.
 - For instance, clustering can be seen as a form of autoassociation.
- There are also neural architectures that learn by autoassociation, but are not autoencoders.
 - Example: Hopfield network

Autoassociation with a Hopfield Network

Original work: J. Hopfield, 1970s

- Hopfield network: a fully connected recurrent network (fully connected graph)
 - Units with 0/1 activations (*unit step function*, Heaviside)
 - An n -unit network requires n^2 weights.
- The input propagates through feedback until convergence
 - Convergence requires symmetric weight matrix and zero self-connections (zero diagonal of the weight matrix)

Hopfield, J. J., Neural networks and physical systems with emergent collective computational abilities, Proceedings of the National Academy of Sciences. 79 (8): 2554–2558, 1982.

Autoassociation with a Hopfield Network

Application of Hopfield Network in Grayscale Image Recognition

Li Tu, Zhiping Liao, and Chi Zhang

https://link.springer.com/chapter/10.1007/978-3-642-27945-4_26

Units correspond to pixels.

Input images:



Autoassociation with a Hopfield Network

Restored images:



(a) Damaged images(Gaussian white noise with mean 0 and variance 0.05)

(b) Restored image



(a) Damaged images (salt and pepper noise with density of 0.4)

(b) Restored image

Limitations: poor generalization; sensitive to translations of images.

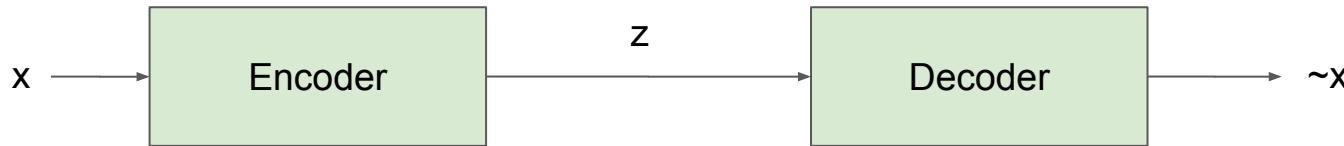
Back to autoencoders

What is an autoencoder?

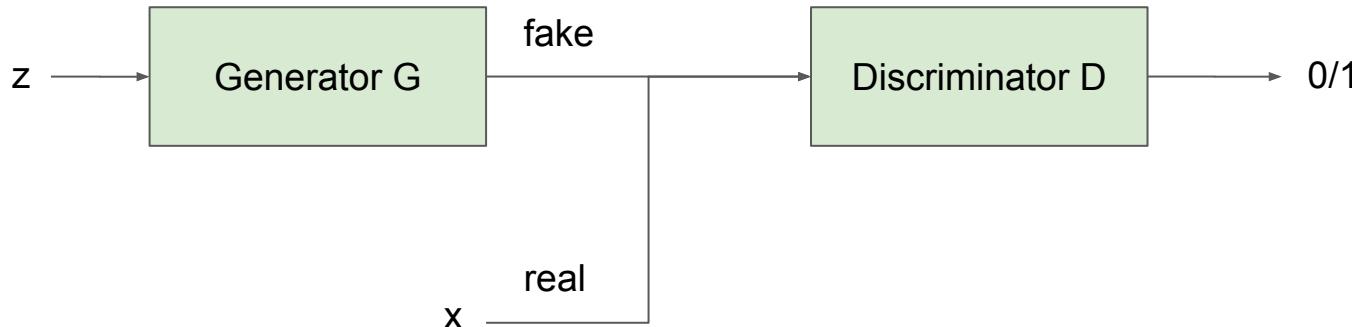
- Composition of encoder $f: X \rightarrow Z$ and decoder $g: Z \rightarrow X$, where $Z \neq X$.
 - f and g can be any architectures: dense, convolutional, recurrent, ...
 - Z imposes a constraint, so that $g \circ f$ “cannot learn to copy perfectly” (I. Goodfellow, Bengio, and Courville, 2016)
- Sounds trivial, but is meant to apply also to scenarios when:
 - x is corrupt in some way, e.g. by
 - noise (in particular more challenging types of noise, like black & pepper in images)
 - ‘occlusion’ (some elements of x unknown or set to default values, e.g. 0)
 - The model is expected to generalize beyond the training sample.
- Note: the terms ‘encoder’ and ‘decoder’ are used in DL more often, but quite often $g \circ f: X \rightarrow Y$, i.e. the output space Y is different than the input space X .

Relation to generative adversarial networks?

AE:



GAN:



The role of the decoder in AE is similar to the role of generator in GANs.

Why autoassociation?

- A form of unsupervised learning.
 - No labels needed: training data easier to obtain; no risk of potential biases incurred by the human labeling.
- A characteristic (defining) feature: x s are composite.
 - e.g. images *composed of* pixels, sequences *composed of* tokens, graphs *composed of* nodes.
- A philosophical argument: to reconstruct x well in a nontrivial way, an AE has to have a good ‘understanding’ of the process that generated x .
- The learning task is harder (more variables/data to predict), so the likelihood of overfitting is lower.
- Richer training signal: each component x_i generates a specific, detailed contribution to the loss function – and therefore to the gradient.

All right, but is that useful in practice?

Usage scenarios

‘Repairing’/recovering autoencoder:

- Denoising
- E.g. U-Net: the latent is the vector of features for an effective receptive field
 - Notice however that U-Net’s architecture does not rely exclusively on the latent.

Clustering:

- When X has too many dimensions to perform sensible/efficient clustering in it.
- Also: Anomaly detection

Task-independent feature synthesis/engineering (representation learning):

- The trained encoder provides a highly informative (close to lossless) encoding of inputs.
 - Can be transplanted into other models (pretraining + fine-tuning)
- May require Z to have certain properties.

Loss function

- *Reconstruction loss, reconstruction error*
- Typically $L(x, g(f(x)))$, where L is a distance metric (or divergence)
- When $X=R^n$, $L=L_2$, and both f and g are linear, the autoencoder learns a form of principal component analysis (PCA), where:
 - f projects x onto a lower dimensional space (reduces dimensionality)
 - g projects $f(x)$ back to X .
 - This can be shown analytically.

Regularizing autoencoders

- For *sparsity* of the latent representation

$$L(x, g(f(x))) + \lambda \sum_i |z_i|$$

- Note: penalization for signal, not for weights!

- For *smoothness* of the latent representation

$$L(x, g(f(x))) + \lambda \|\nabla_x z\|_2$$

- Note: derivative of z w.r.t. x , not w.r.t. weights!
 - Abrupt changes of z (as a function of changes of x) are penalized

- Via intentional *corruption* of the input (denoising autoencoders)

$$L(x, g(f(\tilde{x})))$$

- Sometimes the corrupted data is already given (e.g. noisy image + averaged image)
 - Interesting variant: Noise2Noise <https://arxiv.org/abs/1803.04189>

Limitations of conventional autoencoders

- Even if the dimensionality of Z is lower than of X, Z still has infinite capacity.
 - Even a single real number has infinite capacity
 - (Not true for their floating-point approximations, obviously).
- If f and g are expressible enough, the autoencoder can map X to Z in an arbitrarily complex way.

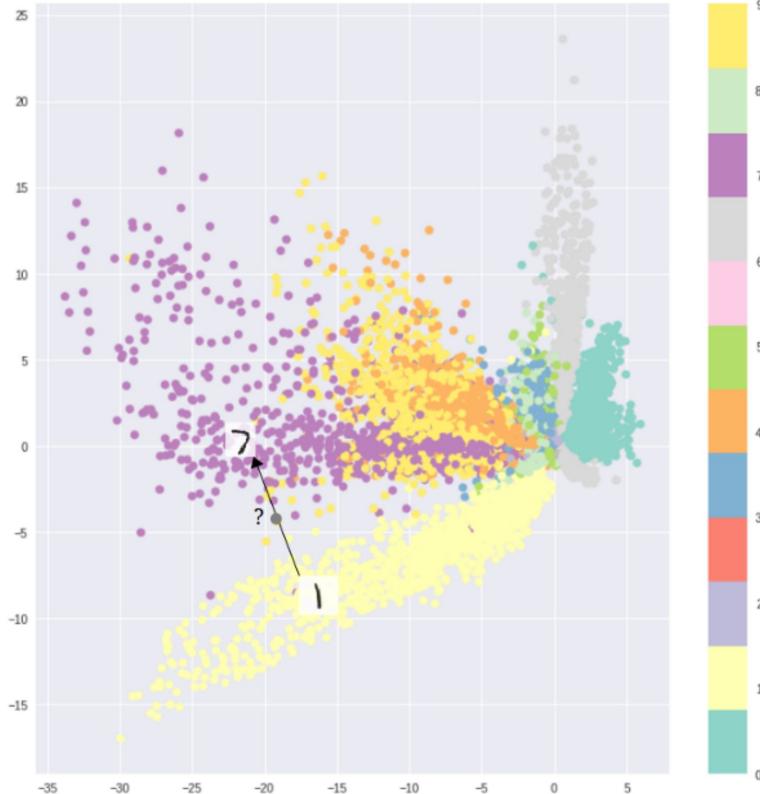
Implications:

- Poor generalization
 - Question: What do we mean by generalization in autoassociative learning?
 - For $x_1, x_2 \in X$, querying g on the points on the segment $(f(x_1), f(x_2))$ in Z may produce unrealistic outputs that do not resemble x_1 nor x_2 .
- Interdependence of dimensions in Z. Implications:
 - Variables in Z are not ‘disentangled’
 - Limited interpretability of Z
 - No easy way of using g as a generator. Very important!

Example

The latent space of an autoencoder projected to 2D using principal component analysis (PCA).

- MNIST dataset: images representing individual digits tend to group nicely in the latent space.
 - ... even though the loss function does not use the labels.
 - Emergent clustering.
- However, interpolating between points in the latent space can produce ‘oddballs’.



Variational Autoencoders

Auto-Encoding Variational Bayes

Diederik P Kingma, Max Welling

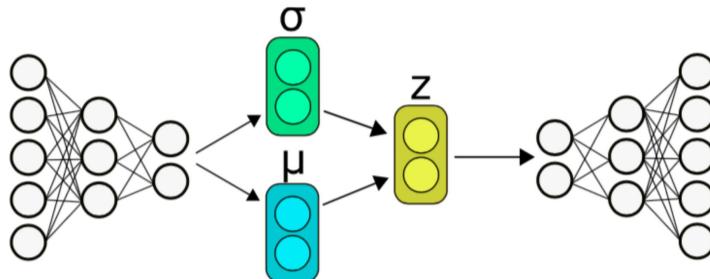
<https://arxiv.org/abs/1312.6114>

Addresses main weaknesses of AEs:

1. Infinite capacity of the latent
The means: making AEs stochastic.
2. Lack of structure in the latent
The means: imposing a specific distribution on the latent.

Making AE stochastic

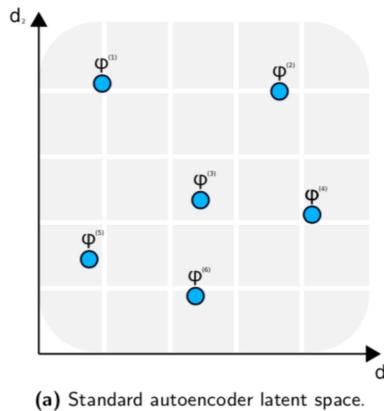
- The values obtained at the latent layer are not directly fed into the decoder g .
- Rather than that, they *parameterize a probability distribution* from which samples are drawn and fed into g .
- The typical distribution of choice: normal distribution.
 - The encoder produces μ and σ , and z is drawn from $N(\mu, \sigma)$.



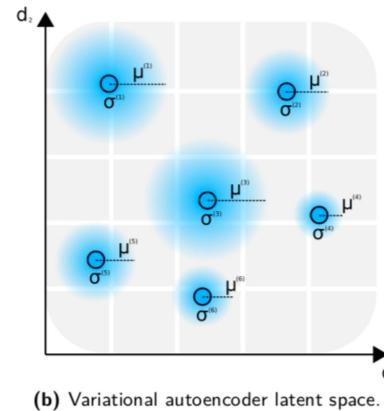
Notice: To produce an n -dimensional latent, we need an encoder with $2n$ outputs.

Making AE stochastic: Implications

Individual samples (x) are mapped to distributions in Z , rather than to points:



(a) Standard autoencoder latent space.



(b) Variational autoencoder latent space.

Technically, requires so-called reparametrization trick, because gradient cannot be propagated through randomness.

- Rather than this:

$$z \sim N(\mu, \sigma)$$

- Use this:

$$z \sim \mu + \sigma \odot N(0, 1)$$

(\odot is elementwise product)

Reparameterization

Requirement: $\sigma \geq 0$

- Can be met by learning $\log(\sigma)$ and then applying $\exp()$

Example: in TensorFlow:

```
def reparameterize(self, mean, logvar):  
    eps = tf.random.normal(shape=mean.shape)  
    return eps * tf.exp(logvar * .5) + mean
```

$$z \sim \mu + \sigma \odot N(0, 1)$$

<https://www.tensorflow.org/tutorials/generative/cvae>

Note: \logvar is $\log(\sigma)$; 0.5 to ensure correspondence between σ and σ^2

Q: Does adding stochasticity solve our problems?

A: No. The stochastic perturbation alone is insufficient.

- The encoder can still ‘stretch’ the latent arbitrarily far.

Enforcing distribution on Z

- Add a loss term that penalizes Z from diverging from some desired distribution.
- In practice, the reference/target distribution is $N_n(0,1)$, where n is the dimensionality of the latent.
 - (n often dropped in formulas, assumed to be inferable from the context)
- Notice: this can be seen as yet another form of *regularization*.
 - We impose additional constraints on the model.

Theoretical formulation

Our aim is to infer the latent variables (posterior) z from the data (evidence) x , i.e.

$$p(z|x)$$

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

but calculating $p(x)$ is intractable:

$$p(x) = \int p(x|z)p(z) dz$$

i.e. we cannot integrate over all possible z , especially when z is high-dimensional.

- A common problem in Bayesian statistics.
 - An alternative could be sampling/Monte Carlo, but it's *inexact* - though has the advantage of being *unbiased*, in contrast to variational inference.

So we try to approximate $p(z|x)$ with $q(z|x)$ by minimizing KL divergence:

$$\min KL(q(z|x) || p(z|x))$$

KL divergence and its relation to entropy

Entropy: expected length of the ‘message’:

$$H(v) = - \sum_i p(v_i) \log p(v_i)$$

Kullback-Leibler divergence:

$$KL(A \parallel B) = \sum_i p_A(v_i) \log \frac{p_A(v_i)}{p_B(v_i)}$$

Equivalent to:

$$KL(A \parallel B) = \sum_i p_A(v_i) \log p_A(v_i) - p_A(v_i) \log p_B(v_i)$$

i.e. the difference of $-H(A)$ and the expected length of B in terms of distribution A.

The latter is the cross entropy $H(A,B)$.

Usually:

- A: data/observations
- B: theoretical/reference distribution.

KL divergence and its relation to entropy

Therefore the relation is:

$$KL(A \parallel B) = -H(A) + H(A, B)$$

Equivalent to:

$$H(A, B) = KL(A \parallel B) + H(A)$$

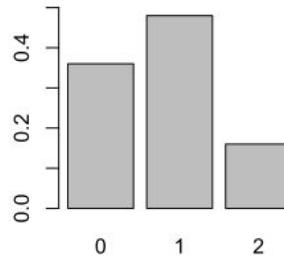
Therefore:

- If $H(A)$ is constant, minimizing $KL(A||B)$ is equivalent to minimizing the cross-entropy $H(A, B)$.
- Notice however that in many practical settings, $H(A)$ is *not* constant, so the above equivalence does not hold.
 - In such cases, minimizing $KL(A||B)$ is equivalent to minimizing the cross-entropy $H(A, B)$ *and simultaneous maximization of $H(A)$* .

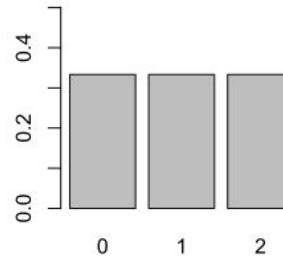
KLD Example

The KL divergence is asymmetric.

Distribution P
Binomial with $p = 0.4$, $N = 2$



Distribution Q
Uniform with $p = 1/3$



x	0	1	2
Distribution P(x)	9/25	12/25	4/25
Distribution Q(x)	1/3	1/3	1/3

$$\text{KL}(P \parallel Q) \approx 0.085$$

$$\text{KL}(Q \parallel P) \approx 0.097$$

Theoretical formulation: key contribution

The authors have shown that

$$\min KL(q(z|x) || p(z|x))$$

can be achieved by maximizing so-call *variational bound*:

$$E_{q(z|x)} \log p(x|z) - KL(q(z|x) || p(z))$$

Note:

- The absence of $p(z|x)$ in the formula.
- $-KL(\dots)$, so this KL divergence is effectively also minimized.
- $p(z)$ is known (we assume p to be $N(0,1)$)
- $p(x|z)$ can be obtained by querying the decoder on the outputs produced by the encoder $(z|x)$ and calculating the cross-entropy of its output w.r.t. input x .
- ELBO = Evidence Lower Bound Optimization

Values vs. probabilities

Important warning: Don't identify the *values* processed/produced by a network with *probabilities (or probability distributions)*.

- This is true in *some* settings: e.g. in classification tasks.
- However, the KL term in VAE architectures expresses requirements with respect to probability distribution.
 - The values produced by the encoder, both before sampling (μ, σ) and after sampling (z), are 'regular' variables, not probabilities on their own.

Implementation (for x being 2D images)

Variational bound (VB): $E_{q(z|x)} \log p(x|z) - KL(q(z|x) || p(z))$

Given that: $KL(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right) = \sum_{x \in \mathcal{X}} P(x) (\log P(x) - \log Q(x))$

The VB is equivalent to:

$$E_{q(z|x)} \log p(x|z) - E_{q(z|x)} (\log q(z|x) - \log p(z))$$

Implementation in TensorFlow:

```
def compute_loss(model, x):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3]) # reduction over image dimensions and channels
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)
    return -tf.reduce_mean(logpx_z + logpz - logqz_x) # compare to the last formula above
```

`log_normal_pdf(x, m, logvar)` calculates the logarithm of the PDF of $N(m, \exp(\text{logvar}))$ at x

Relationship with reconstruction error

- The first term is essentially the (negated) reconstruction error
 - Even though we didn't even ask for that!

$$E_{q(z|x)} \log p(x|z)$$

- One may think of $p(x|z)$ as $p(x|x^\wedge)$, because the decoder produces x^\wedge from z deterministically: $x^\wedge = g(z)$.
- Assume $p(x|x^\wedge)$ is Gaussian; then it contains the term

$$e^{-|x - \hat{x}|^2}$$

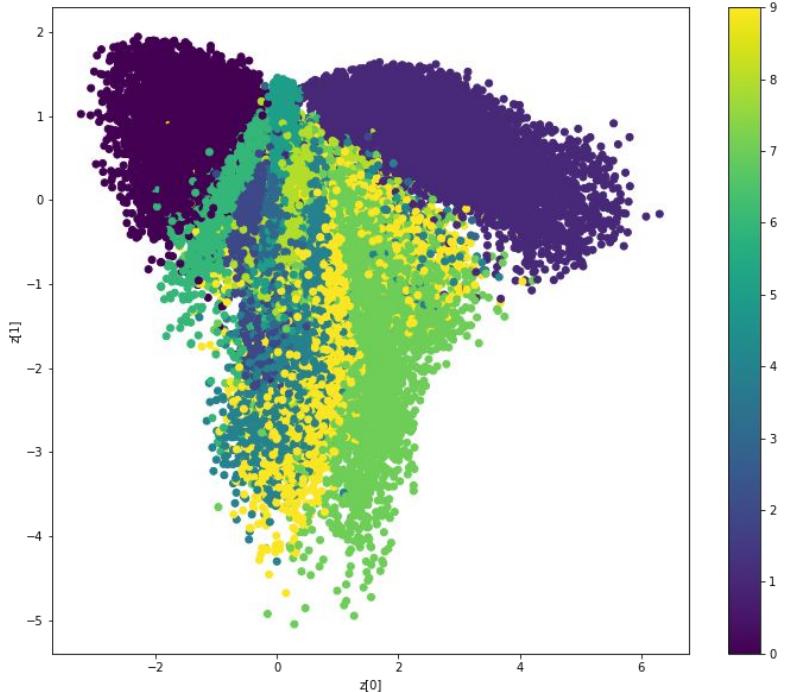
which after logarithm becomes $-|x - x^\wedge|^2$ – negated square error.

Of course $p(x|x^\wedge)$ is not guaranteed in general to be Gaussian, but it's likely to be Gaussian-like, given (among others) the Gaussian sampling in the latent.

Latent space of a VAE for MNIST

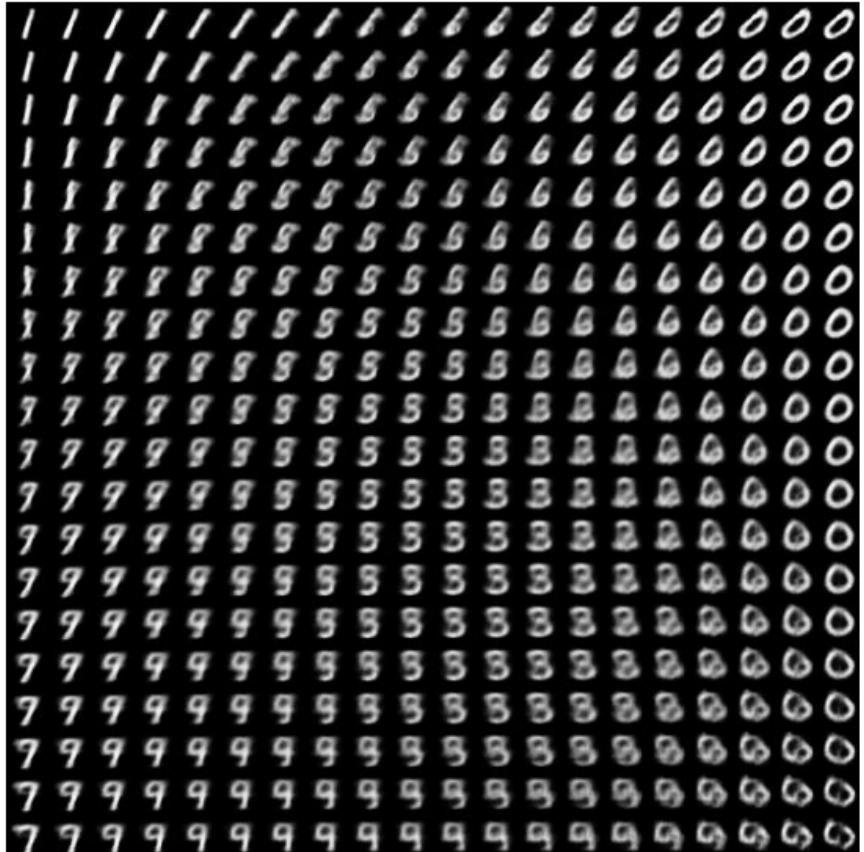
Compare to the latent space of a regular AE:

- Compact, no empty spaces
- Roughly centered around (0,0)



Interpretability of the latent

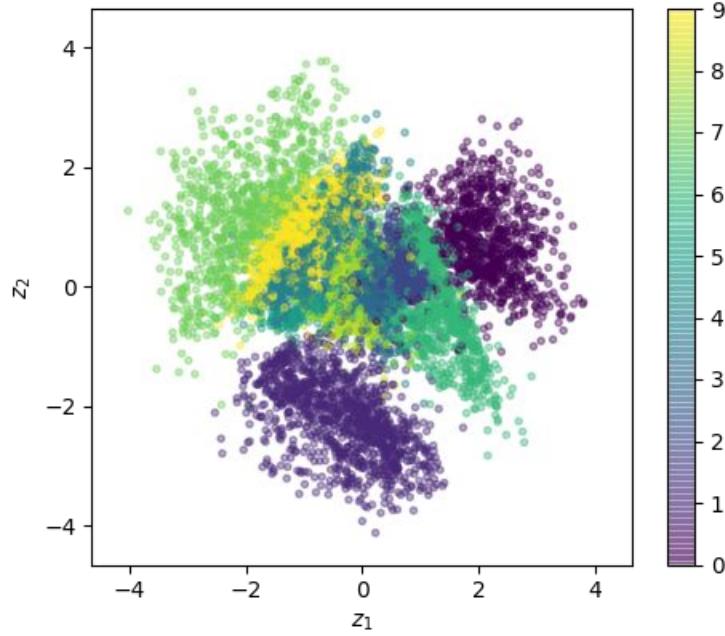
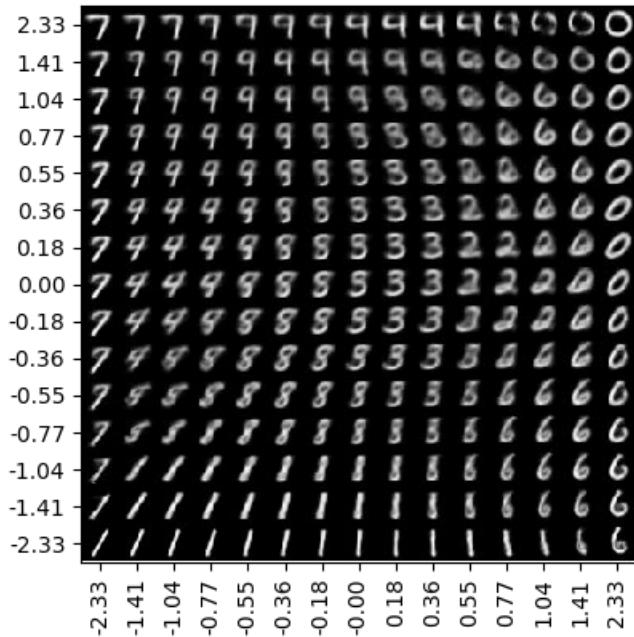
- For 2D latent space
- Grid locations correspond to z_1 and z_2 iterating over quantiles of $N(0,1)$ in interval (0.05, 0.95)
- Each digit/image is the output of the decoder queried on the corresponding latent vector.
- We observe gradual transition of digits' shapes.
- Almost all images look realistic.
- In some cases, variables can be attributed with some 'semantics'.



Another visualization

Left: decoder's responses to sweeping the latent space.

Right: the latent space with MNIST decision classes labeled.



Implications of the isotropic latent

- The target distribution for the latent space (typically $N(0,1)$) is isotropic.
 - Diagonal covariance matrix.
- The outputs of the encoder need to be as independent as possible.
 - Any dependence of latent dimensions causes loss of isotropy, and will be penalized by the KL divergence term in the loss function.
- The encoder is thus trained to disentangle the features/characteristics (conceptual ‘dimensions’) of the inputs.
 - For instance for handwritten characters: a separate dimension for the slant.
 - Popular term: disentangled representations.
- The discovered dimensions can, but don’t have to correspond to human concepts and percepts; this depends on:
 - What are the main factors of variance in the inputs (X)?
 - How correlated they are?
 - Can they be captured using the specific architecture of the encoder? (e.g. convolutional NN)

Notes

A more general loss function (this time minimized):

$$\mathcal{L}(x, \hat{x}) + \sum_j KL(q_j(z|x) || p(z))$$

where L is *some* form of reconstruction error (e.g. MSE).

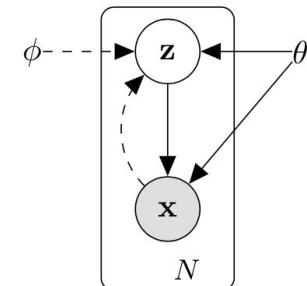
Compare with the formula derived by the authors: $E_{q(z|x)} \log p(x|z) - KL(q(z|x) || p(z))$

Note:

- Watch the units: \mathcal{L} may be not expressed in nats/bits anymore.
 - This may require weighing these two terms accordingly.
- Here q is factored by its dimensions
 - This is however unnecessary if $p(z)$ is Gaussian, as dimensions in $N(0,1)$ are also Gaussians - $N(0,1)$ is isotropic.

Notes

- Imposing distribution on the latent is a form of regularization.
- Imposing *convenient* distribution on the latent facilitates using decoder as a generator.
 - We can easily draw pseudorandom numbers from $N(0,1)$.
- Variational Inference = techniques for approximating complex probability distributions.
- VAEs are probabilistic graphical models.
- VAEs perform generative modeling:
they attempt to simulate how the data is generated,
in order to understand the underlying causal relations.
 - Note that predictive/discriminative modeling is ‘just’
about learning predictors given observations.



About the dimensionality of the latent

What is the ‘right’ dimensionality for the latent space, given a distribution of examples in X? (in practice: a training set)

The related concept: intrinsic dimension

In the fields of pattern recognition and machine learning the intrinsic dimension for a data set can be thought of as the number of variables needed in a minimal representation of the data. Similarly, in signal processing of multidimensional signals, the intrinsic dimension of the signal describes how many variables are needed to generate a good approximation of the signal.

https://en.wikipedia.org/wiki/Intrinsic_dimension

Analogs to intrinsic dimension exist elsewhere; e.g. intrinsic dimension of an order.

https://en.wikipedia.org/wiki/Order_dimension

Reiterated/summary: Why variational autoencoders?

Advantage 1:

- In a regular autoencoder (AE), all we have is just a mapping $X \rightarrow Z \rightarrow X$
- The distribution in Z is unknown.
 - Its *closed form* is unknown.
 - The decoder cannot be used alone as a generator.
- In a variational autoencoder (VAE), we demand the distribution in Z to approximate a distribution given in a closed form.
 - Now the decoder can be used as a generator.

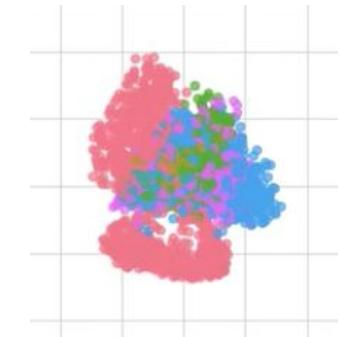
Advantage 2:

- In general, better generalization, thanks to the latent space being more compact.

Dynamics of VAE training

Nice illustration of the ‘evolution’ of the latent space during training of a VAE on the MNIST dataset:

- Notice how the initial mapping realized by the encoder produces distribution that is:
 - Not isotropic.
 - Not centered at the origin of the coordinate system.
- With the progress of training, this distribution becomes more isotropic and close to centered at (0,0).



https://giphy.com/gifs/vae-lqq0em9cuivVNWFwSX?utm_source=iframe&utm_medium=embed&utm_campaign=Embeds&utm_term=https%3A%2F%2Fjaan.io%2F

Wasserstein Auto-Encoders

Wasserstein Auto-Encoders

Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, Bernhard Schoelkopf

<https://arxiv.org/abs/1711.01558>

Wasserstein Auto-Encoders

Similarly to Wasserstein GANs, the idea is to use a different way of measuring the divergence from probability distribution in the latent from the target distribution.

- In other words: different way of *regularizing* the model.
- Consequence: better quality of generated samples.

Other types of autoencoders

For Kullback-Leibler divergence (KL) as the measure of discrepancy between the actual distribution of the latent and the desired distribution, the corresponding models are variational autoencoders (VAEs).

When choosing the maximum-mean discrepancy (MMD), we obtain Wasserstein autoencoders (WAES).

- A WAE is an AE with a different regularizer than ‘regular’ VAEs.
- The main advantage of WAES over VAEs is that MMD allows using encoders with deterministic outputs, while, by definition, KL requires only encoders with stochastic outputs.
- MMD should be preferred over KL, when using deterministic encoders.

WAE vs. VAE

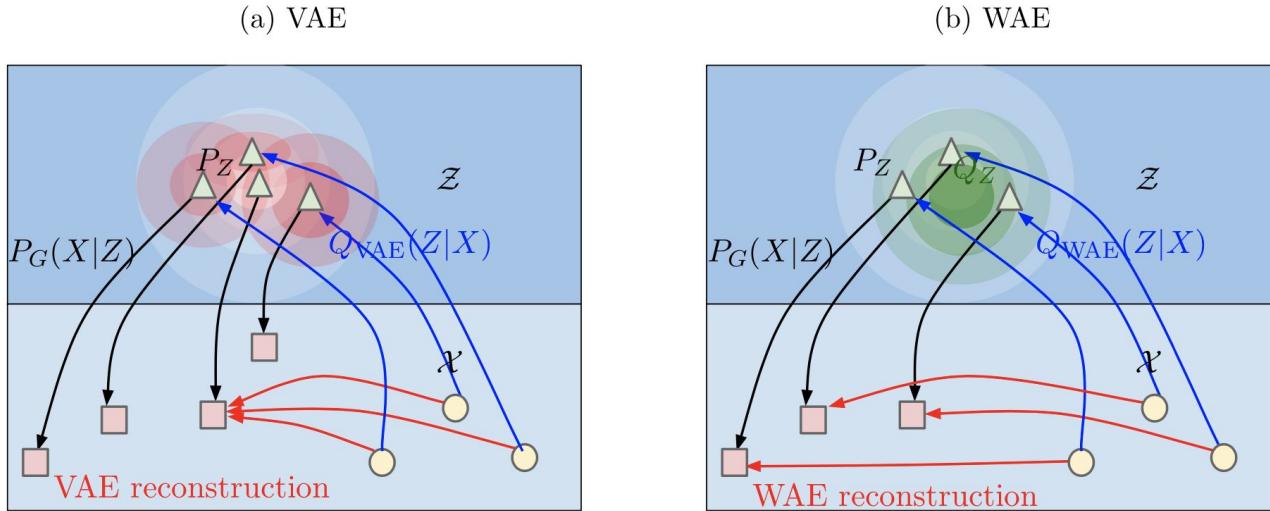


Figure 1: Both VAE and WAE minimize two terms: the reconstruction cost and the regularizer penalizing discrepancy between P_Z and distribution induced by the encoder Q . VAE forces $Q(Z|X = x)$ to match P_Z for all the different input examples x drawn from P_X . This is illustrated on picture (a), where every single red ball is forced to match P_Z depicted as the white shape. Red balls start intersecting, which leads to problems with reconstruction. In contrast, WAE forces the continuous mixture $Q_Z := \int Q(Z|X)dP_X$ to match P_Z , as depicted with the green ball in picture (b). As a result latent codes of different examples get a chance to stay far away from each other, promoting a better reconstruction.

Regularizers

The authors propose two regularizers: GAN-based and MMD-based (maximum mean discrepancy).

Algorithm 1 Wasserstein Auto-Encoder with GAN-based penalty (WAE-GAN).

Require: Regularization coefficient $\lambda > 0$.

Initialize the parameters of the encoder Q_ϕ , decoder G_θ , and latent discriminator D_γ .

while (ϕ, θ) not converged **do**

 Sample $\{x_1, \dots, x_n\}$ from the training set

 Sample $\{z_1, \dots, z_n\}$ from the prior P_Z

 Sample \tilde{z}_i from $Q_\phi(Z|x_i)$ for $i = 1, \dots, n$

 Update D_γ by ascending:

$$\frac{\lambda}{n} \sum_{i=1}^n \log D_\gamma(z_i) + \log(1 - D_\gamma(\tilde{z}_i))$$

 Update Q_ϕ and G_θ by descending:

$$\frac{1}{n} \sum_{i=1}^n c(x_i, G_\theta(\tilde{z}_i)) - \lambda \cdot \log D_\gamma(\tilde{z}_i)$$

end while

Algorithm 2 Wasserstein Auto-Encoder with MMD-based penalty (WAE-MMD).

Require: Regularization coefficient $\lambda > 0$,

characteristic positive-definite kernel k .

Initialize the parameters of the encoder Q_ϕ , decoder G_θ , and latent discriminator D_γ .

while (ϕ, θ) not converged **do**

 Sample $\{x_1, \dots, x_n\}$ from the training set

 Sample $\{z_1, \dots, z_n\}$ from the prior P_Z

 Sample \tilde{z}_i from $Q_\phi(Z|x_i)$ for $i = 1, \dots, n$

 Update Q_ϕ and G_θ by descending:

$$\begin{aligned} & \frac{1}{n} \sum_{i=1}^n c(x_i, G_\theta(\tilde{z}_i)) + \frac{\lambda}{n(n-1)} \sum_{\ell \neq j} k(z_\ell, z_j) \\ & + \frac{\lambda}{n(n-1)} \sum_{\ell \neq j} k(\tilde{z}_\ell, \tilde{z}_j) - \frac{2\lambda}{n^2} \sum_{\ell, j} k(z_\ell, \tilde{z}_j) \end{aligned}$$

end while

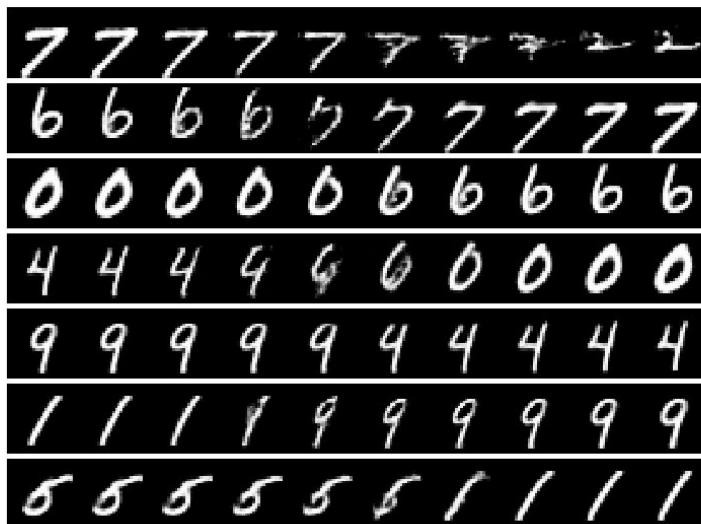
Results

Experimented with MNIST and CelebA dataset.

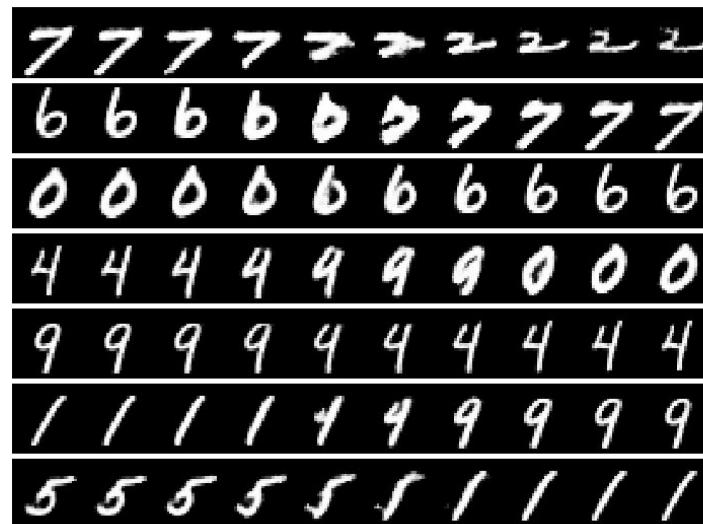
Exemplary interpolations (first and last image a reconstructions of examples from the test set).

Test interpolations

VAE



WAE-MMD



Related reading and materials

- Understanding Entanglement With SVD

<https://www.math3ma.com/blog/understanding-entanglement-with-svd>

Module 4: Recurrent Neural Networks

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Outline

1. Recurrent Architectures
2. LSTM: Where it all started (more or less)
 - a. Notable variants (including GRU)
3. Examples of RNN usage
 - a. The Seq2seq blueprint
4. An example of follow-up of the Seq2Seq paradigm
5. Related concepts
6. Orthogonal initialization

Recurrent Architectures

Recurrent vs. recursive

recurrent | rɪ'kʌr(ə)nt |

- occurring often or repeatedly: she had a recurrent dream about falling.

recursive | rɪ'kə:sɪv |

characterized by recurrence or repetition.

- *Mathematics & Linguistics* relating to or involving the repeated application of a rule, definition, or procedure to successive results: this restriction ensures that the grammar is recursive.
- *Computing* relating to or involving a program or routine of which a part requires the application of the whole, so that its explicit interpretation requires in general many successive executions: a recursive subroutine.

Recurrent vs. recursive

A rule of thumb:

- ‘recurrence’ is a property of data (a sequence of events),
- ‘recursion’ is a property of a model, algorithm, etc.

Most RNN architectures

- can learn about recurrence of signals/events over time,
and
- are recursive, because the computation at each step is conducted using the
same function, which is being applied to the outcomes of the previous
iteration.

Learning from sequences

Generic finite sequences:

- $x^0, x^1, \dots, x^t, \dots, x^T$

where x^t are usually vectors (multidimensional time series).

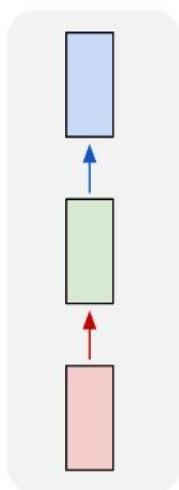
- (but RNNs can be easily applied to other data types, by mapping the ‘raw’ sequence elements to vectors, and vice versa for outputs)

Key properties:

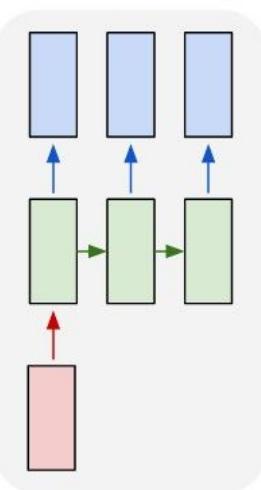
- Variable length
- Ordering of elements does matter.
 - No permutation invariance (if present, were dealing with *sets*, not sequences).
 - Structure/pattern(s) exist along the time axis
- Inference requires some form of *state* and *state accumulation*.
 - Many recurrent NNs are *stateful* in this sense, to address this need.
 - (Some RNNs, particularly the early ones, were not stateful).

Usage scenarios

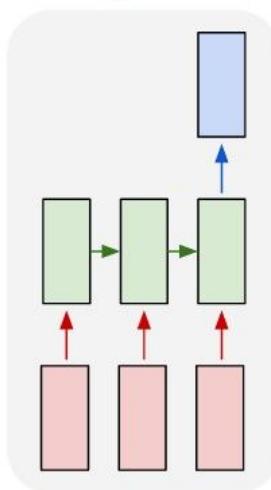
one to one



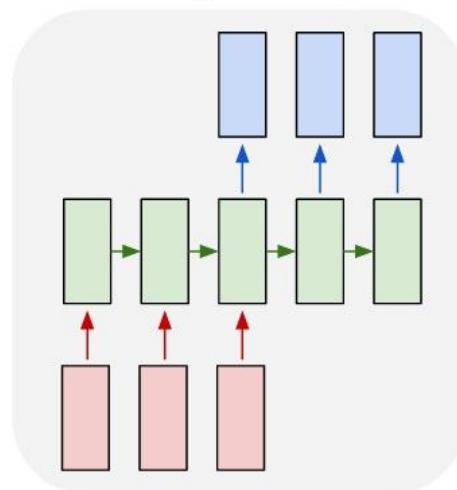
one to many



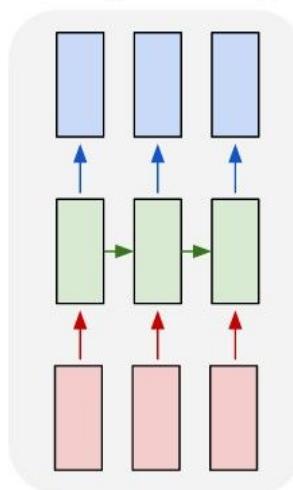
many to one



many to many



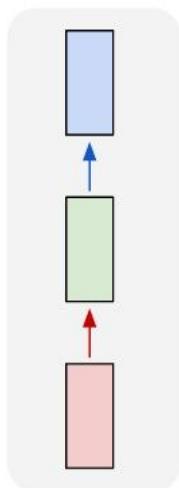
many to many



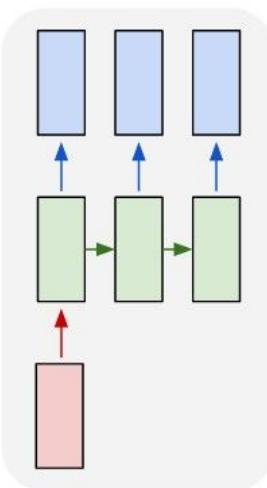
Usage scenarios

Relationship to concepts known from functional programming:

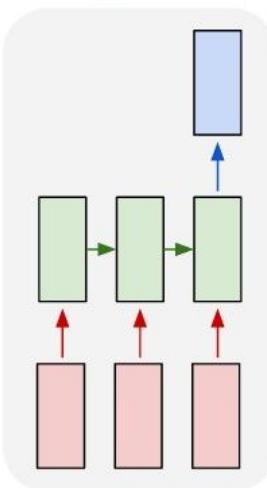
one to one



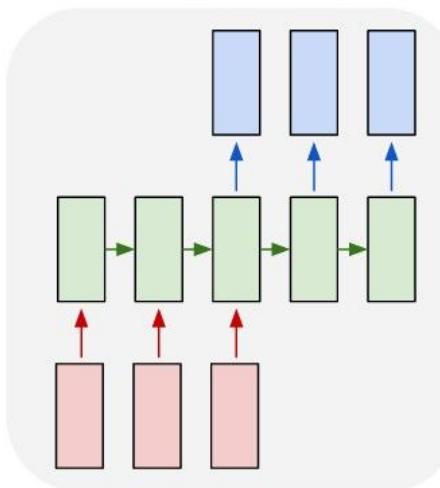
one to many



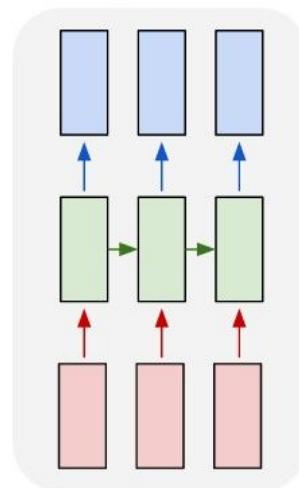
many to one



many to many



many to many



unfold

fold

fold+unfold

“map”

Main challenges in learning time sequences

Simple recurrent NNs:

- Were stateless (a memory-less function).
- Were difficult to train.
- Had problems when modeling long time dependencies.

Core idea:

- Maintain state.
 - Note: all models considered so far in this course were stateless (timeless).
- ‘Self-control’ by introducing the concept of gating signals.

Long short-term memory (LSTM)

Long short-term memory

Sepp Hochreiter; Jürgen Schmidhuber

Neural Computation, 9 (8), 1997: 1735–1780.

<https://dl.acm.org/doi/10.1162/neco.1997.9.8.1735>

Following description based on:

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2222–2232.

<https://arxiv.org/abs/1503.04069>

2020 vs. 2019



Juergen Schmidhuber

FOLLOW

The Swiss AI Lab IDSIA / USI & SUPSI
Verified email at idsia.ch - [Homepage](#)

TITLE	CITED BY	YEAR
Long short-term memory S Hochreiter, J Schmidhuber Neural computation 9 (8), 1735-1780	25836	1997



Juergen Schmidhuber

FOLLOW

The Swiss AI Lab IDSIA / USI & SUPSI
Verified email at idsia.ch - [Homepage](#)
[computer science](#) [artificial intelligence](#) [reinforcement learning](#) [neural networks](#) [physics](#)

TITLE	CITED BY	YEAR
Long short-term memory S Hochreiter, J Schmidhuber Neural computation 9 (8), 1735-1780	40939	1997

'Vanilla' LSTM

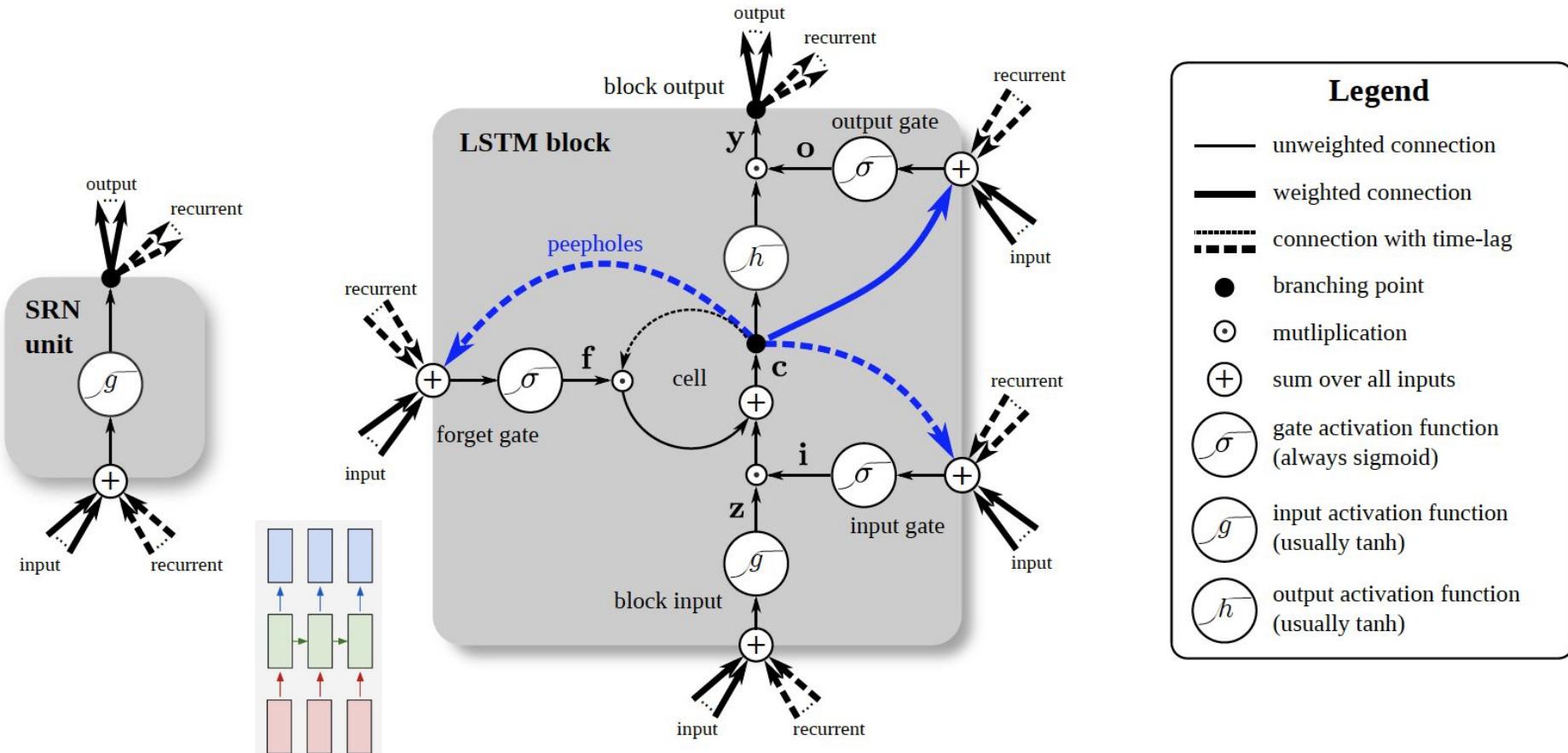
Core components:

- State: a vector
- Three gates:
 - input gate: decides how much the current input influences the state,
 - forget gate: can reset the state of the cell (“soft” reset, selective by state dimension),
 - output gate: decides how much of the current state should be passed to the next layer.
- Peephole connections [optional]

Notes:

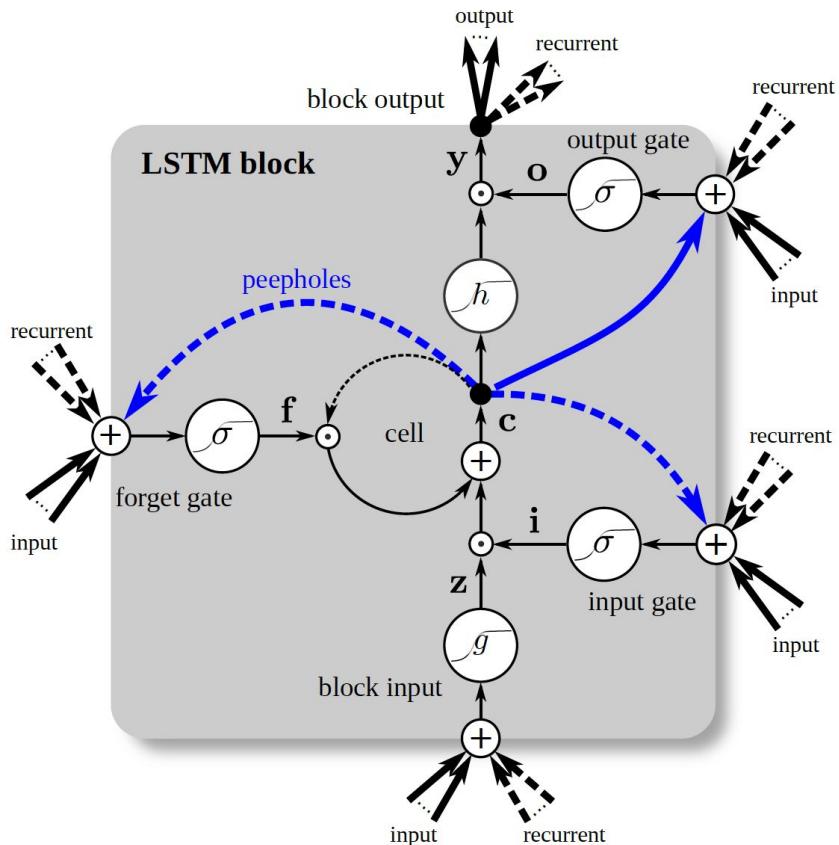
- The processing corresponding to a single element (dimension) of the state vector is sometimes referred to as *LSTM cell*.
- The output y goes back to input x and to all the gates.
- The description that follows is slightly upgraded compared to the 1997 version.

Stateless recurrent unit vs. ‘vanilla’ LSTM





Details on signal propagation



Input x^t : M-dimensional vector

$$\bar{z}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z$$

$$z^t = g(\bar{z}^t)$$

$$\bar{i}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i$$

$$i^t = \sigma(\bar{i}^t)$$

$$\bar{f}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f$$

$$f^t = \sigma(\bar{f}^t)$$

$$c^t = z^t \odot i^t + c^{t-1} \odot f^t$$

$$\bar{o}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o$$

$$o^t = \sigma(\bar{o}^t)$$

$$y^t = h(c^t) \odot o^t$$

block input

input gate

forget gate

cell

output gate

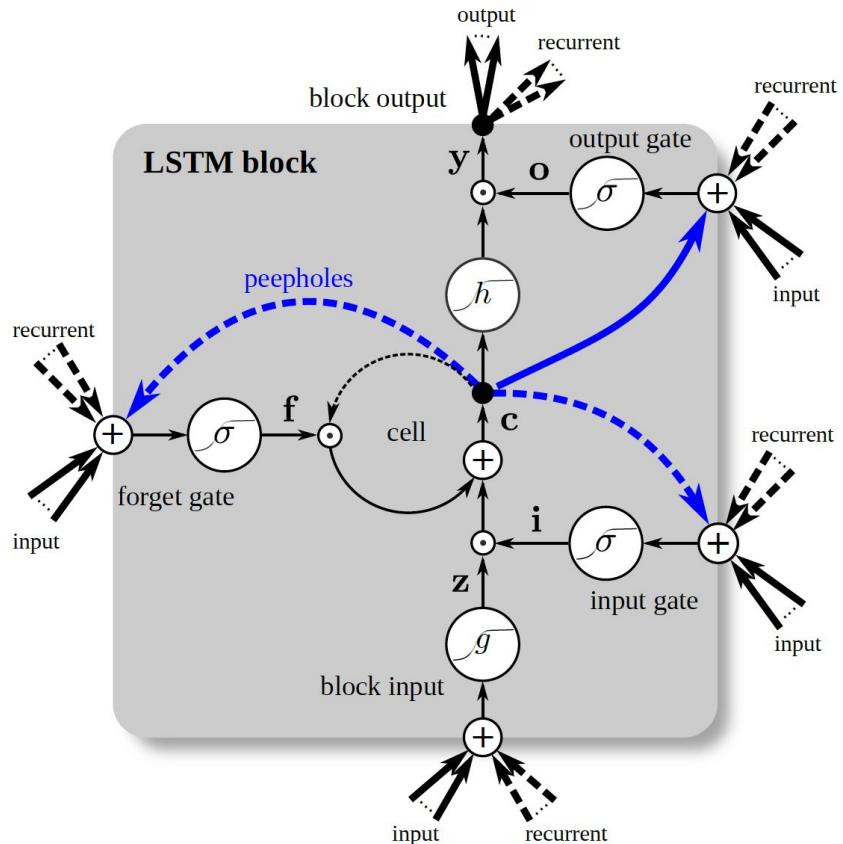
block output

Notice the repeating pattern $W_*x^t + R_*y^{t-1}$

Parameters

- M: number of *inputs* (input ‘channels’) in x,
- N: *internal* dimensionality,
 - also the number of outputs from the layer, i.e. dimensionality of y,
- Weight matrices:
 - Input weights: $\mathbf{W}_z, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o \in \mathbb{R}^{N \times M}$
 - Recurrent weights: $\mathbf{R}_z, \mathbf{R}_i, \mathbf{R}_f, \mathbf{R}_o \in \mathbb{R}^{N \times N}$
 - Peephole weights: $\mathbf{p}_i, \mathbf{p}_f, \mathbf{p}_o \in \mathbb{R}^N$
 - Bias weights: $\mathbf{b}_z, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^N$
- Implications:
 - All internal quantities and the output are vectors of length N
 - For a single LSTM block (N=1), internal quantities are scalars
 - ‘Mixing’ of inputs ($M \rightarrow N$ dimensions) occurs only at the entries.
 - For a sequence of T elements of dimensionality M ($T \times M$ tensor), a LSTM cell produces a sequence of T elements of dimensionality N ($T \times N$ tensor)
 - Independently of that, batching is typically possible too (e.g. in TensorFlow)

Training: Backpropagation through time



Δ^t : the vector of deltas passed down from the layer 'above' (next layer/component)

$$\delta \mathbf{y}^t = \Delta^t + \mathbf{R}_z^T \delta \mathbf{z}^{t+1} + \mathbf{R}_i^T \delta \mathbf{i}^{t+1} + \mathbf{R}_f^T \delta \mathbf{f}^{t+1} + \mathbf{R}_o^T \delta \mathbf{o}^{t+1}$$

$$\delta \bar{\mathbf{o}}^t = \delta \mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\bar{\mathbf{o}}^t)$$

$$\begin{aligned} \delta \mathbf{c}^t &= \delta \mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \mathbf{p}_o \odot \delta \bar{\mathbf{o}}^t + \mathbf{p}_i \odot \delta \bar{\mathbf{i}}^{t+1} \\ &\quad + \mathbf{p}_f \odot \delta \bar{\mathbf{f}}^{t+1} + \delta \mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \end{aligned}$$

$$\delta \bar{\mathbf{f}}^t = \delta \mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\bar{\mathbf{f}}^t)$$

$$\delta \bar{\mathbf{i}}^t = \delta \mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\bar{\mathbf{i}}^t)$$

$$\delta \bar{\mathbf{z}}^t = \delta \mathbf{c}^t \odot \mathbf{i}^t \odot g'(\bar{\mathbf{z}}^t)$$

Deltas for the inputs (to be backpropagated to the preceding layer):

$$\delta \mathbf{x}^t = \mathbf{W}_z^T \delta \bar{\mathbf{z}}^t + \mathbf{W}_i^T \delta \bar{\mathbf{i}}^t + \mathbf{W}_f^T \delta \bar{\mathbf{f}}^t + \mathbf{W}_o^T \delta \bar{\mathbf{o}}^t$$

Training: Backpropagation through time

$$\bar{\mathbf{z}}^t = \mathbf{W}_z \mathbf{x}^t + \mathbf{R}_z \mathbf{y}^{t-1} + \mathbf{b}_z$$

$$\mathbf{z}^t = g(\bar{\mathbf{z}}^t)$$

$$\bar{\mathbf{i}}^t = \mathbf{W}_i \mathbf{x}^t + \mathbf{R}_i \mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i$$

$$\mathbf{i}^t = \sigma(\bar{\mathbf{i}}^t)$$

$$\bar{\mathbf{f}}^t = \mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f$$

$$\mathbf{f}^t = \sigma(\bar{\mathbf{f}}^t)$$

$$\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$$

$$\bar{\mathbf{o}}^t = \mathbf{W}_o \mathbf{x}^t + \mathbf{R}_o \mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o$$

$$\mathbf{o}^t = \sigma(\bar{\mathbf{o}}^t)$$

$$\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t$$

block input

input gate

forget gate

cell

output gate

block output

Δ^t : the vector of deltas passed down from the layer ‘above’ (next layer/component)

$$\delta \mathbf{y}^t = \Delta^t + \mathbf{R}_z^T \delta \mathbf{z}^{t+1} + \mathbf{R}_i^T \delta \mathbf{i}^{t+1} + \mathbf{R}_f^T \delta \mathbf{f}^{t+1} + \mathbf{R}_o^T \delta \mathbf{o}^{t+1}$$

$$\delta \bar{\mathbf{o}}^t = \delta \mathbf{y}^t \odot h(\mathbf{c}^t) \odot \sigma'(\bar{\mathbf{o}}^t)$$

$$\begin{aligned} \delta \mathbf{c}^t = \delta \mathbf{y}^t \odot \mathbf{o}^t \odot h'(\mathbf{c}^t) + \mathbf{p}_o \odot \delta \bar{\mathbf{o}}^t + \mathbf{p}_i \odot \delta \bar{\mathbf{i}}^{t+1} \\ + \mathbf{p}_f \odot \delta \bar{\mathbf{f}}^{t+1} + \delta \mathbf{c}^{t+1} \odot \mathbf{f}^{t+1} \end{aligned}$$

$$\delta \bar{\mathbf{f}}^t = \delta \mathbf{c}^t \odot \mathbf{c}^{t-1} \odot \sigma'(\bar{\mathbf{f}}^t)$$

$$\delta \bar{\mathbf{i}}^t = \delta \mathbf{c}^t \odot \mathbf{z}^t \odot \sigma'(\bar{\mathbf{i}}^t)$$

$$\delta \bar{\mathbf{z}}^t = \delta \mathbf{c}^t \odot \mathbf{i}^t \odot g'(\bar{\mathbf{z}}^t)$$

Deltas for the inputs (to be backpropagated to the preceding layer):

$$\delta \mathbf{x}^t = \mathbf{W}_z^T \delta \bar{\mathbf{z}}^t + \mathbf{W}_i^T \delta \bar{\mathbf{i}}^t + \mathbf{W}_f^T \delta \bar{\mathbf{f}}^t + \mathbf{W}_o^T \delta \bar{\mathbf{o}}^t$$

Training: Backpropagation through time

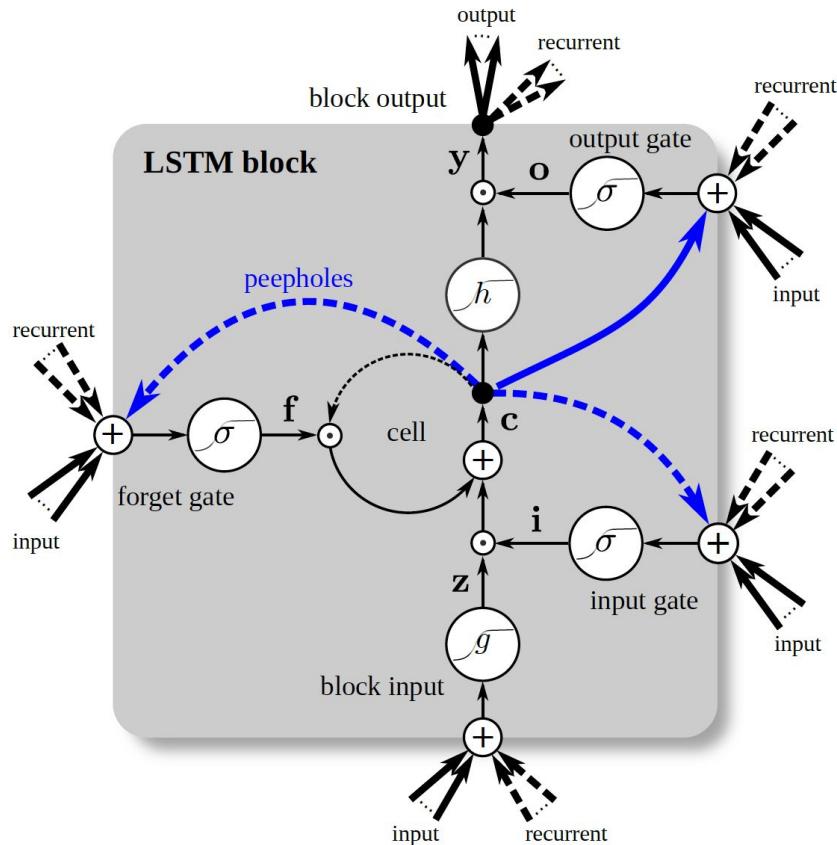
The previous slide shows the calculation of gradients for a single iteration.

- How to perform training for an entire sequence of length T?
- The final step: gradients aggregated over the considered time horizon [0,T]:

$$\begin{aligned}\delta \mathbf{W}_\star &= \sum_{t=0}^T \langle \delta \star^t, \mathbf{x}^t \rangle & \delta \mathbf{p}_i &= \sum_{t=0}^{T-1} \mathbf{c}^t \odot \delta \bar{\mathbf{i}}^{t+1} \\ \delta \mathbf{R}_\star &= \sum_{t=0}^{T-1} \langle \delta \star^{t+1}, \mathbf{y}^t \rangle & \delta \mathbf{p}_f &= \sum_{t=0}^{T-1} \mathbf{c}^t \odot \delta \bar{\mathbf{f}}^{t+1} \\ \delta \mathbf{b}_\star &= \sum_{t=0}^T \delta \star^t & \delta \mathbf{p}_o &= \sum_{t=0}^T \mathbf{c}^t \odot \delta \bar{\mathbf{o}}^t\end{aligned}$$

where star denotes any of the internal quantities (before being passed through nonlinearity, i.e. those with dashes); $\langle \rangle$ is scalar product.

Observations



LSTM features two levels of recursion:

- internal, aimed at maintaining the state c ,
- external, resulting from y being fed back to the cell in each iteration.

Peephole connections: optional; meant to allow the state to control the gates ‘immediately’, i.e. in the same step.

- Without them, the control always lags by one time step.

LSTM variants

“Ablated” variants considered in the paper:

- NIG: No Input Gate: $i^t = 1$
- NFG: No Forget Gate: $f^t = 1$
- NOG: No Output Gate: $o^t = 1$
- NIAF: No Input Activation Function: $g(x) = x$
- NOAF: No Output Activation Function: $h(x) = x$
- CIFG: Coupled Input and Forget Gate: $f^t = 1 - i^t$

More complex ones:

- NP: No Peepholes
- FGR: Full Gate Recurrence

The empirical assessment conducted in the paper demonstrates that the full ('vanilla') variant of LSTM performs best in most cases.

Demos shown in the paper ('Space odyssey')

- TIMIT: speech recognition
 - Input: 12 MFCCs (Mel Frequency Cepstrum Coefficients) + spectral energy
 - Task: classification of phonemes
- JSB Chorales:
 - Inputs: MIDI sequences of 382 JS Bach chorales transposed to C major or C minor, sampled every quarter note
 - Task: next-step prediction
 - Loss function: minimizing neg log-likelihood
 - A sample (other authors): <https://www.youtube.com/watch?v=Iz8xQou2OqA>
- IAM Online: The IAM Online Handwriting Database
 - (see next slide)

IAM Online: example

IAM Online Handwriting Database

- Input: pen trajectory
 - A sequence of tuples (Δx , Δy , t, up/down)
 - No information on character start/end in the input sequence
- Output: sequence of characters

Example = a single line of text.

Ben Zoma said: "The days of 1thy life means in the day-time; all the days of 1thy life means even at night-time."
(Berochoth.) And the Rabbis thought it important that when we read the

(a)

Ben Zoma said: "The days of 1thy life means in the day-time; all the days of 1thy life means even at night-time."
(Berochoth.) And the Rabbis thought it important that when we read the

(b)

Gated Recurrent Unit

Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

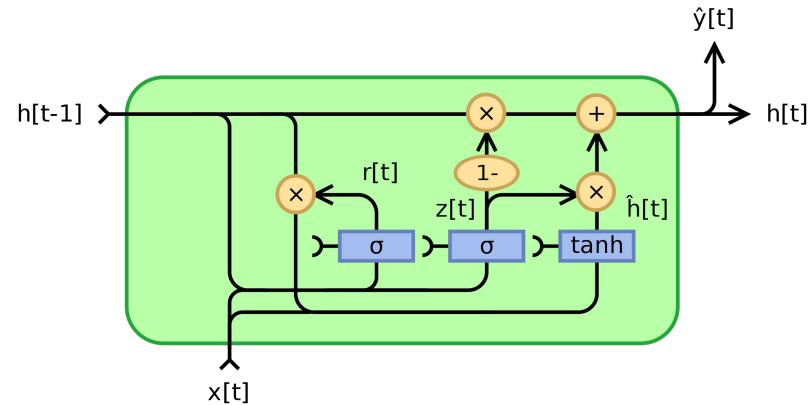
Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio

<https://arxiv.org/abs/1412.3555>

Gated Recurrent Unit

Flow diagram (a different, quite popular style of diagrammatic representation):

- σ : sigmoid activation function
- h : state,
- \times : marks gating (elementwise/pointwise/Hadamard product)
- \circ - marks parameters, i.e. multiplication by weight matrix before passing through a given activation function.



Key differences w.r.t. LSTM:

- no peephole connections,
- no output activation function,
- the input and forget gate coupled into an update gate,
 - z combines the mix of the previous state $h[t-1]$ and the current update $\hat{h}[t]$

GRUs vs. LSTM

- GRU has fewer parameters than LSTM.
 - Easier training.
- GRU's performance on certain tasks (polyphonic music modeling, speech signal modeling, and more) was found to be similar to that of LSTM.
- GRUs have been shown to exhibit even better performance on certain smaller datasets.
 - Perhaps because fewer parameters mean less effort per epoch, and thus more epochs within the same budget?

However:

- LSTM is "strictly stronger" than the GRU as it can perform unbounded counting, while the GRU cannot [1]:
 - That is why GRU tends to fail to learn in certain domains that are learnable by the LSTM.

[1] Weiss, Goldberg, Yahav, On the Practical Computational Power of Finite Precision RNNs for Language Recognition, 2018,
<https://arxiv.org/abs/1805.04908>

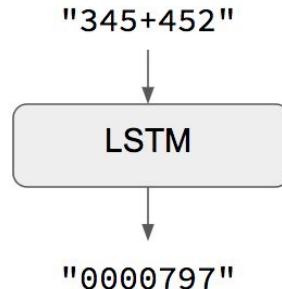
Other observations concerning RNNs

- Can be applied to any sequences, not only time sequences
- Can be used in bi-directional mode
 - Technically: a pair of LSTM cells: forward, backward.
- Fare pretty well on sequences that are meant to represent nonlinear structures, e.g. trees
 - Example: Expression trees in prefix notation: (+ (* 2 7) (- 3 x))
- Paved the way for recurrent architectures capable of processing non-linear data structures, e.g. trees.
- Don't use RNNs for sets: sets (unordered sequences) are permutation-invariant, and RNNs are designed specifically with sequential nature of data in mind.
 - There are other deep architectures designed to process sets, e.g. Deep Sets, <https://arxiv.org/abs/1703.06114>

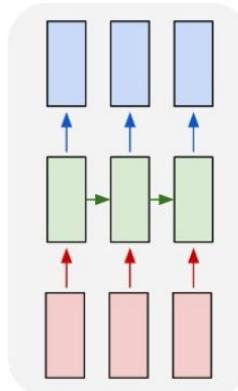
Selected usage patterns of RNNs

The Seq2seq blueprint

- A broad class of tasks that consist in mapping sequences to sequences.
- The simplest variant: mapping of sequences of same length.
- Can be implemented with a single RNN, e.g. LSTM or GRU.
- Example*: training an RNN to add numbers encoded as character strings:



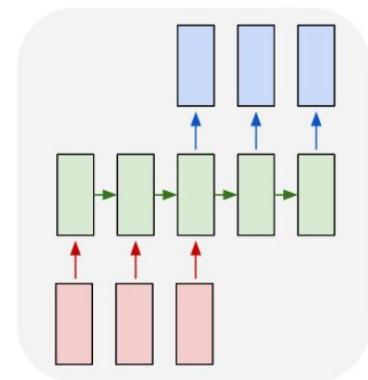
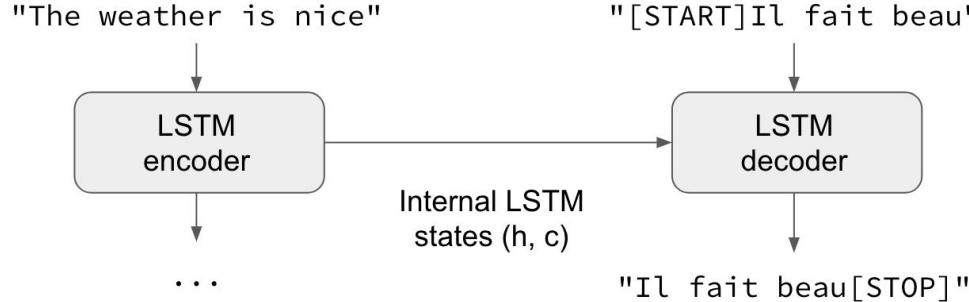
- Notice the padding zeros.



*A ten-minute introduction to sequence-to-sequence learning in Keras, F. Chollet,
<https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html>

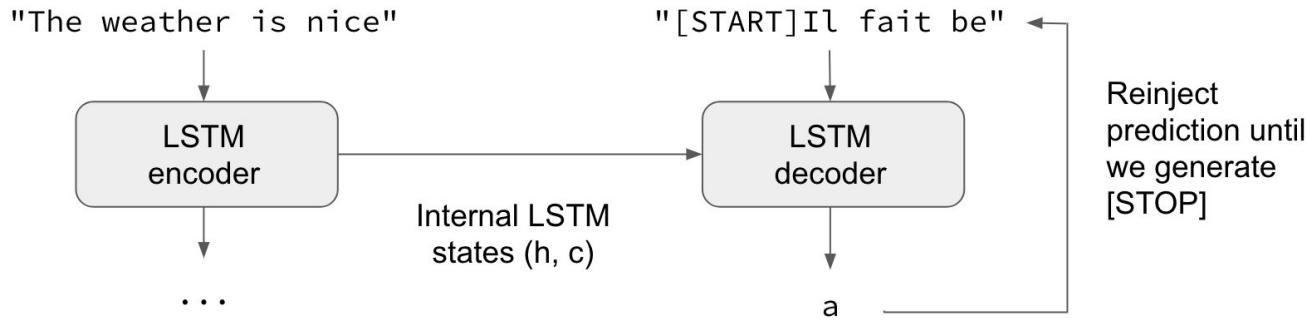
The Seq2seq blueprint

- The case for non-equal lengths: “canonical” seq2seq
- Two RNNs (e.g. LSTMs): encoder and decoder
- Encoder “folds” the input sequence into its hidden state h (i.e. a fixed-length latent representation)
- Decoder’s state is initialized with h and is trained to predict the next token in the target sequence, given the previous tokens of the target sequence as input (it should reproduce it’s input shifted by one token) – so called teacher forcing.



Querying of the Seq2seq model

The consecutive tokens produced by the decoder are fed back to its input.



Notice: this scheme can be also used to train the decoder without teacher forcing.

General remark: In DL, the terms ‘encoder’ and ‘decoder’ have quite general meaning, i.e. are not necessarily meant only to encode from and decode to the same type/space (like in autoencoders), but more generally.

The Seq2seq blueprint

- Used widely in Machine Translation, Text Summarization, Conversational Modeling, and more
 - One of the milestones of Neural Machine Translation (NMT)
- Typically accompanied with word embeddings (token embeddings).
 - Embedding = a mapping from categorical domain to Cartesian space.
 - The basic form of embedding: learnable look-up table of n entries (dictionary size), each containing an (initially random) vector of m reals (embedding dimensionality).
 - Notable representatives: Glove, Word2Vec, FastText, ELMo, ...
- A vast range of variants
 - Often involves additional over-the-sequence attention mechanisms

However, Seq2seq are recently superseded by Transformer models [1].

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin,
Attention Is All You Need, <https://arxiv.org/abs/1706.03762>

Embeddings

The objective: To map the value of an m-valued categorical/nominal/discrete variable to \mathbb{R}^n : $\{1, \dots, m\} \rightarrow \mathbb{R}^n$

Motivations:

1. To address the deficiencies of one-hot encoding, which
 - o becomes inefficient for large m,
 - o is ‘uninformed’ about the ‘semantics’ of the variable.
 - Relations between the values of the variable cannot be reflected in one-hot encoding.
2. To provide continuity of representation (\mathbb{R} in place of {0,1}).

Most embeddings are parametric, and thus trainable*:

1. Training starts from a random parameterization.
2. The parameters of the embedding are updated with gradient.

The trained embedding can reflect the ‘semantics’ of the values.

* Special case: random [key] embedding: non-trainable.

Embeddings

The simplest embeddings are just [learnable] arrays indexed by the categorical variable (lookup tables).

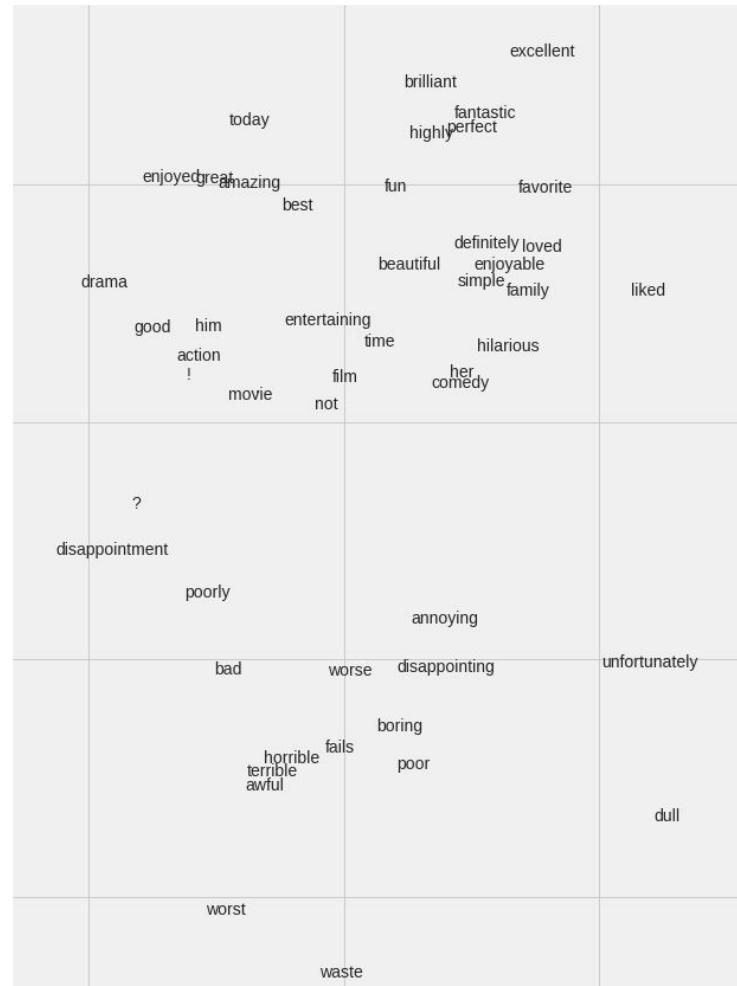
- $f: [1, m] \rightarrow \mathbb{R}^n$
 - (the domain contains all possible values of the variable in some fixed order)
- Implemented as $m \times n$ array.
 - Initialization: random.
 - Querying: simply indexing of the rows of the array: $f(i)$ returns i^{th} row of the array.
 - Training: the back-propagated gradient allows calculating the updates for the i^{th} row.
 - The remaining rows remain intact.
- As such, lookup tables are not differentiable.
 - Indexing (choice) is a non-differentiable operation.
 - This is however unproblematic in most cases, as embeddings usually serve as interfaces between the categorical input variables and the model.
 - There is no need to backpropagate gradient through them.

Embedding: Example

A fragment of a $n=100$ -dimensional embedding of $m=50,000$ -size vocabulary of words used in movie reviews.

- Trained as a part of a model that predicts the sentimentality of reviews.
- Projected on the 2D plane.

Words of similar sentiment clearly cluster together.



Source:

https://colab.research.google.com/notebooks/mlcc/intro_to_sparse_data_and_embeddings.ipynb

Embeddings

Natural language-related embeddings: rely on the domain knowledge specific for NLP and concrete language. Examples:

- Word embeddings
- Sentence embeddings
- More sophisticated embeddings involve relationships between the elements of NL (words, phrases, sentences)

Recommended reading

On the right: exemplary fragments of Shakespeare-style plays/dramas, generated using a 3-layer RNN (a rather unsophisticated model by today's standards).

PANDARUS:

Alas, I think he shall be come approached and the day
When little strain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:

They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:

Well, your wit is in the care of side and that.

Second Lord:

They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:

Come, sir, I will make did behold your worship.

VIOLA:

I'll drink it.

Andrej Karpathy blog
The Unreasonable Effectiveness of Recurrent Neural Networks,
May 21, 2015

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

Other examples of applications of RNNs

- Mapping (translation of) images of math formulas to Latex code
 - <https://mathpix.com/>
- Simplifying/transforming mathematical expressions

For the example on the right:

$i \hbar \frac{d}{dt} |\Psi(l)\rangle = \hat{H} |\Psi(l)\rangle$

See e.g.:

Guillaume Lample, François Charton, Deep Learning for Symbolic Mathematics,
<https://arxiv.org/abs/1912.01412>

Equation [edit]

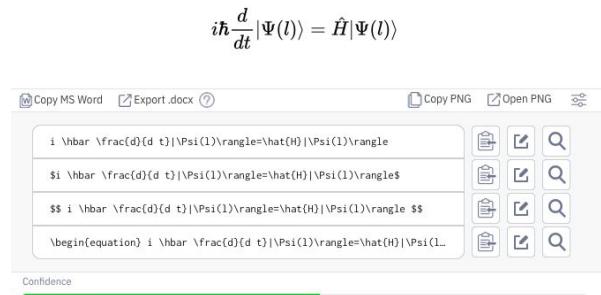
Time-dependent equation [edit]

The form of the Schrödinger equation depends on the physical situation. The time-dependent Schrödinger equation (TDSE), which gives a description of a system

Time-dependent Schrödinger equation (general)

$$i\hbar \frac{d}{dt} |\Psi(t)\rangle = \hat{H} |\Psi(t)\rangle$$

where i is the imaginary unit, $\hbar = \frac{h}{2\pi}$ is the reduced Planck constant, Ψ (the Greek letter psi) is the state vector of the quantum system, \hat{H} is the Hamiltonian operator. The position-space wave function of the quantum system is the expansion of the state vector in terms of the position eigenstates $|\Psi(r,t)\rangle = \langle r|\Psi\rangle$. Similarly, the momentum-space wave function



An example of follow-up of the
Seq2Seq paradigm

Tree2tree (recurrent) autoencoder

Ain't Nobody Got Time for Coding: Structure-Aware Program Synthesis from Natural Language

Jakub Bednarek, Karol Piaskowski, Krzysztof Krawiec

<https://arxiv.org/abs/1810.09717>

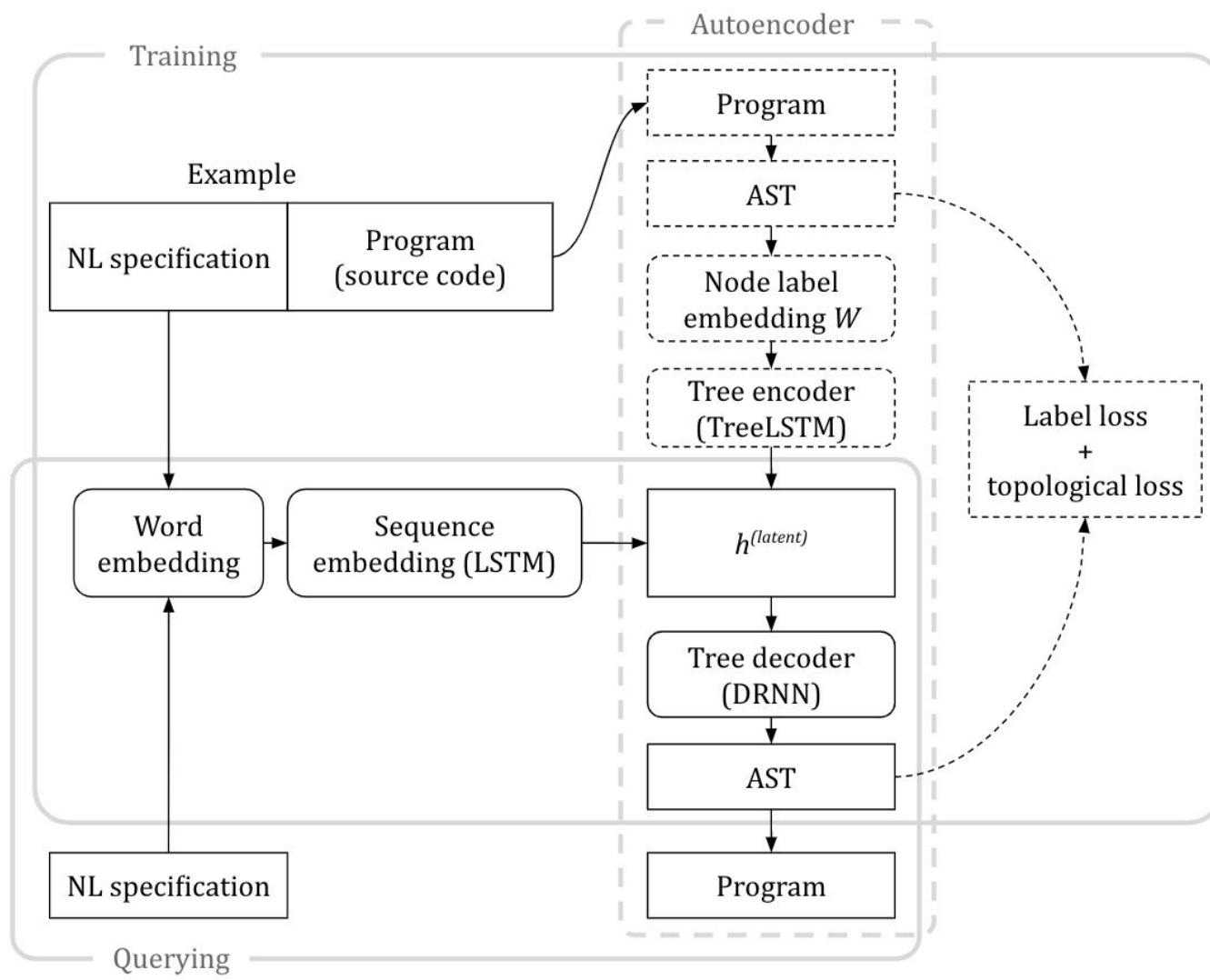
Motivations

Program synthesis: automatic generation of programs from specifications.

Forms of specifications used traditionally in for synthesis:

- examples of desired program behavior (input-output pairs),
- formal specifications (logical clauses, typically so-called *preconditions* and *postconditions*, a.k.a. *contracts*),
- partial programs (*sketching*).

What is the most natural form of specification from the human viewpoint?



The benchmark (Polosukhin & Skidanov, 2018)

- Functional language based on Lisp (AlgoLisp)
- Three types: string, bool, function
- The grammar:

```
program      ::= symbol
symbol       ::= constant | argument | function_call | function | lambda
constant     ::= number | string | True | False
function_call ::= (function_name arguments)
function     ::= function_name
arguments    ::= symbol | arguments , symbol
function_name ::= Reduce | Filter | Hap | Head | + | - | ...
lambda       ::= lambda function_call
```

- An example of an (input, output) pair:

You are given an array a. Find the smallest element in a, which is strictly greater than the minimum element in a

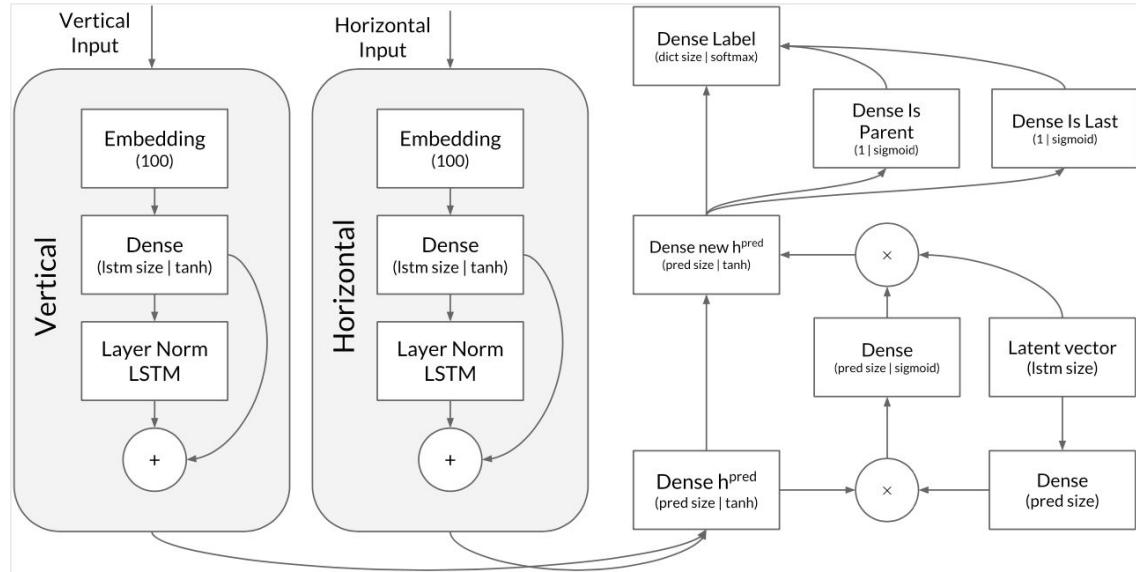
```
(reduce (filter a (partial0 (reduce a inf) <)) inf min)
```

- 99506 examples in total, split into training (79214) validation (9352) and test set (10940 examples).

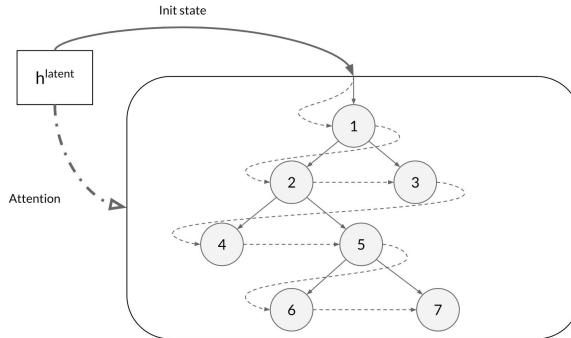
The decoder

Doubly-recurrent NN: a RNN nested inside an RNN.

- The outer RNN iterates over tree levels
- The inner RNN iterates over nodes within a level



Order of traversal of the nodes in the output tree:



Some results

Percentage of perfectly synthesized programs (i.e., syntactically identical to the target ones) for SAPS configurations trained from scratch:

Model	Validation	Test	Number of parameters [M]
SAPS V	0%	0%	5.06
SAPS V Att	0%	0%	5.17
SAPS H	84.31%	78.92%	5.06
SAPS H Att	89.54%	86.20%	5.17
SAPS VH	93.65%	89.10%	5.62
SAPS VH Att	93.73%	92.36%	5.73

Comparison with other models:

Model	Validation	Test
Attentional Seq2Seq	54.4%	54.1%
Seq2Tree	61.2%	61.0%
SAPSpre VH Att (256)	86.67%	83.80%
Seq2Tree + Search	86.1%	85.8%

Notice the good performance of Seq2Seq!

Examples of synthesized programs

Specification	Synthesized program
<p>you are given numbers a and b, your task is to find $a + b$</p> <p>you given numbers a b, your is find $a + b$</p> <p>given a numbers b, find $a + b$</p> <p>given a numbers b, $a + b$</p> <p>a b, $a + b$</p>	$(+, \ a, \ b)$ $(+, \ a, \ b)$ $(+, \ a, \ b)$ $(+, \ a, \ b)$ $(+, \ (+, \ a, \ b), \ c)$
<p>you are given numbers a and b, your task is to find a multiplied by b</p> <p>you are given numbers a and b, your task is to find minimum a and b</p>	$(\star, \ a, \ b)$ $(\min, \ a, \ b)$
<p>given a number a and an array of numbers b, find the length of the longest subsequence of range from 0 to a inclusive that is a prefix of b</p> <p>given a number a and an array of numbers b, find the length of the longest subsequence of range from 1 to a exclusive that is a prefix of b</p>	$(\text{reduce}, \ (\text{range}, \ 0, \ (+, \ a, \ 1)), \ 0,$ $(\text{lambda2}, \ (\text{if}, \ (==, \ \text{arg2}, \ (\text{if}, \ (<, \$ $\text{arg1}, \ (\text{len}, \ b)), \ (\text{deref}, \ b, \ \text{arg1}), \$ $0)), \ (+, \ \text{arg1}, \ 1), \ \text{arg1}))$ $(\text{reduce}, \ (\text{range}, \ 1, \ a), \ 0, \ (\text{lambda2},$ $(\text{if}, \ (==, \ \text{arg2}, \ (\text{if}, \ (<, \ \text{arg1}, \ (\text{len}, \$ $b)), \ (\text{deref}, \ b, \ \text{arg1}), \ 0)), \ (+, \ \text{arg1}, \$ $1), \ \text{arg1}))$
<p>given an array of numbers a, find median of values in a after only keeping first half</p> <p>given an array of numbers a, find mean of values in a after only keeping second half</p>	$(\text{deref}, \ (\text{sort}, \ (\text{slice}, \ a, \ 0, \ (/,$ $(\text{len}, \ a), \ 2))), \ (/, \ (\text{len}, \ (\text{slice}, \ a,$ $0, \ (/, \ (\text{len}, \ a), \ 2))), \ 2))$ $(/, \ (\text{reduce}, \ (\text{slice}, \ a, \ (/, \ (\text{len}, \ a),$ $2), \ (\text{len}, \ a)), \ 0, \ +), \ (\text{len}, \ (\text{slice},$ $a, \ (/, \ (\text{len}, \ a), \ 2), \ (\text{len}, \ a))))$

Recurrent Neural Networks: Related concepts

Technical limitations

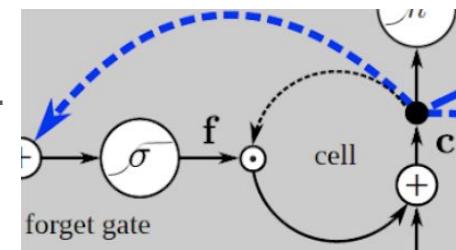
- Processing in RNNs is strictly sequential, and as such does not lend itself to parallelization.
 - Processing time in general proportional to the length of the sequence.
 - Implication: RNNs can be more efficient on CPUs than on GPUs
- A (partial) remedy: batch processing
 - Typical convention of tensor dimensions: (batch, time, channels)
 - Some frameworks enable swapping the positions of batch and time on demand, as it speeds up processing a bit (e.g. the time_major option in TensorFlow).
- A challenge: sequences of different lengths cannot be forced into a rectangular tensor (equal size on all dimensions required)
- Solutions:
 - Grouping sequences of same length in training and forming batches of them
 - Padding sequences with <null> tokens (may somehow disturb the training)
 - Using specialized data structures, e.g. RaggedTensor in TensorFlow.

Signal gating vs. attention

Gating used in RNNs can be likened to an attention mechanism.

Similarities:

- Both involve elementwise product \odot of the signal and the gate output



Differences:

- Gating applies to individual signals (channels).
- Attention is (typically) applied over other dimensions:
 - space (e.g. in convolutional networks in image analysis)
 - time

Gating is most similar to soft attention: the signals are weighed by coefficients from (0,1), so no channel is entirely silenced.

- Hard attention involves ‘strict’ selection of elements (e.g. cropping of rasters). 53

Risks incurred by gating

What happens to gradient at \odot operator?

- \odot is simply the scalar product applied to each element
- For a single channel (c-signal, f-gate output):

$$(cf)' = c'f + cf'$$

- f is constrained to $(0,1)$, but c not (in general)
- Therefore, cf' can get arbitrarily large (arbitrary c , f' constrained only by the steepness of the sigmoidal activation function)
- This may destabilize the training process
 - Vanishing and exploding gradients are common for RNNs.
- Potentially, this problem applies to any part of a model where signals are multiplied elementwise.
 - For instance attention mechanisms.

Risks incurred by gating: remedies (1)

Gradient clipping

- Clip (clamp) gradient vector to a fixed interval
 - Allows precise definition of upper and lower limit, e.g. in TF:
`tf.clip_by_value(grad, -1., 1.)`
- Applied after all gradients in the computation graph have been calculated.
- Note: This does not make the graph as a whole discontinuous (and thus not differentiable): the gradient has been already calculated.
- Advantage: fast.

Disadvantage:

- Will in most cases change the *direction* of the gradient
- Example: $[3, 0.8] \rightarrow [1, 0.8]$

Risks incurred by gating: remedies (2)

Gradient normalization

- Normalize gradient vector to unit length
 - Example: [3, 4] -> [%, %]
- Applied after all gradients in the graph have been calculated
- Advantage: does not change gradient direction

Disadvantage:

- Computationally more costly
- The dimension of the greatest magnitude may dominate the gradient.
- In extreme cases, may effectively reduce gradient on some dimensions to (almost) zero.
 - Example: [1, 1000] -> [~0.001, 1]
- Clipping and normalization often used together.

Signal regularization

- Note: *signal* regularization, not parameter regularization!
- Enforces/promotes sparsity of signal
- In LSTM, can be applied to the output of a LSTM block.

For instance in the TensorFlow implementation of LSTM:

- Controlled by the `activity_regularizer` parameter.

In addition, TensorFlow facilitates other types of regularization:

- `kernel_regularizer` (weight matrix for inputs, W^*)
- `recurrent_regularizer` (weight matrix for recurrent processing, R^*)
- `bias_regularizer` (bias/offset vector)

Orthogonal initialization

Orthogonal initialization

- Recurrent processing in RNNs involves multiple multiplications of a weight matrix
- This is likely to lead to the matrix ‘vanishing’ or ‘exploding’
- Caveat: The explanation that follows ignores activation functions.
 - We simply assume a weight matrix that is being multiplied by itself multiple times.

Source:

Stephen Merity, Explaining and illustrating orthogonal initialization for recurrent neural networks,

https://smerity.com/articles/2016/orthogonal_init.html

Example: Fibonacci sequence

- The terms of the Fibonacci sequence (and some other recursive sequences) can be calculated via matrix multiplication

$$F^n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} F^{n-1} & F^n \\ F^n & F^{n+1} \end{pmatrix}$$

$$\begin{aligned}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^4 &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^4 \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^3 \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \\ &= \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \\ &= \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}\end{aligned}$$

Example: Fibonacci sequence

F can be factorized as

$$F = Q\Lambda Q^{-1}$$

where Q is the matrix of eigenvectors, and Λ is the matrix of eigenvalues.

For the Fibonacci sequence:

$$Q = \begin{pmatrix} -0.8506 & -0.5257 \\ 0.5257 & -0.8506 \end{pmatrix}$$

$$\Lambda = \begin{pmatrix} -0.6180 & 0 \\ 0 & 1.6180 \end{pmatrix}$$

$$Q^{-1} = \begin{pmatrix} -0.8506 & 0.5257 \\ -0.5257 & -0.8506 \end{pmatrix}$$

Let's try to calculate the second term:

$$\begin{aligned} F^2 &= (Q\Lambda Q^{-1})(Q\Lambda Q^{-1}) \\ &= Q\Lambda(Q^{-1}Q)\Lambda Q^{-1} \\ &= Q\Lambda^2 Q^{-1} \end{aligned}$$

Example: Fibonacci sequence

In general:

$$F^n = Q\Lambda^n Q^{-1}$$

With increasing n, the largest eigenvalue dominates the Λ matrix.

For instance, for n=10:

$$\Lambda^{10} = \begin{pmatrix} 0.0081 & 0 \\ 0 & 122.9919 \end{pmatrix}$$

In general:

- If $|\text{eigenvalues}| < 1 \Rightarrow$ matrix vanishes
- If $|\text{eigenvalues}| > 1 \Rightarrow$ matrix explodes
- If $|\text{eigenvalues}| = 1 \Rightarrow$ matrix maintains a constant norm.

What are the implications for initialization of RNNs?

RNN initialization

Assuming identity activation function f , the hidden state h in step t is:

$$h_t = f(Wh_{t-1} + Vx_t) = f(Wh_{t-1}) = Wh_{t-1}$$
$$h_3 = W(W(Wh_0)) = W^3h_0 = W^3I = W^3$$

If we use W that is orthonormal, i.e.

$$W^T W = WW^T = I,$$

all its eigenvalues are 1, and there is no explosion nor vanishing. This also reduces risk of exploding/vanishing gradient.

Caveat: f is an identity activation. For a more general form of f , the above chain of argument would be much more difficult.

- Nevertheless, notice that for $f=\tanh$, f is close to identity for small arguments.

See https://smerity.com/articles/2016/orthogonal_init.html for visualizations of matrices.

Andrew M. Saxe, James L. McClelland, Surya Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks

<https://arxiv.org/abs/1312.6120>

RNNs that model differential equations

RNNs that model differential equations

Motivations:

- DEs are the lingua franca of physics
- Very well researched, known properties
- Elaborate solvers and other software
- The ‘proper’ kind of models for all sorts of problems and domains, including medicine, economics, industry, etc.

The representative of this thread covered in the next slides:

Liquid Time-constant Networks

Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, Radu Grosu

<http://arxiv.org/abs/2006.04439>

The objective

- Devise a ‘Neural ODE’, a neural model that implements an Ordinary Differential Equation (ODE)

$$d\mathbf{x}(t)/dt = f(\mathbf{x}(t), \mathbf{I}(t), t, \theta),$$

where: \mathbf{x} is the state, \mathbf{I} is the input, θ are the parameters of the network

More stable formulation:

$$\frac{d\mathbf{\tilde{x}}(t)}{dt} = -\frac{\mathbf{x}(t)}{\tau} + f(\mathbf{x}(t), \mathbf{I}(t), t, \theta),$$

The first term on the right-hand side helps the system to reach the equilibrium state with a time constant τ .

Authors' proposition: liquid time-constant

$$\frac{d\mathbf{x}(t)}{dt} = - \left[\frac{1}{\tau} + f(\mathbf{x}(t), \mathbf{I}(t), t, \theta) \right] \mathbf{x}(t) + f(\mathbf{x}(t), \mathbf{I}(t), t, \theta) A.$$

where A is a parameter.

The advantage: the network not only determines the derivative of the hidden state x , but also predicts the input-dependent time-constant (which is technically 1 over the term in the square brackets).

The authors coin this *liquid time-constant recurrent neural networks* (LTCs).

Features of LTCs

- Are differentiable and can be differentiated with reverse-mode autodiff.
- Bounded stability: both the state x and the time constant τ are bounded to a finite range, which assures stability of output dynamics.
- Expressivity:
 - LTCs are universal approximators (proof in the paper).
 - Can be shown to express complex trajectories (see next slides).
- Good empirical performance on a range of tasks.
- Interesting biological motivations:
 - Somewhat related to computational models of neural dynamics in ‘small species’.
 - Resemble Dynamical Causam models (Friston, Harrison, and Penny 2003).

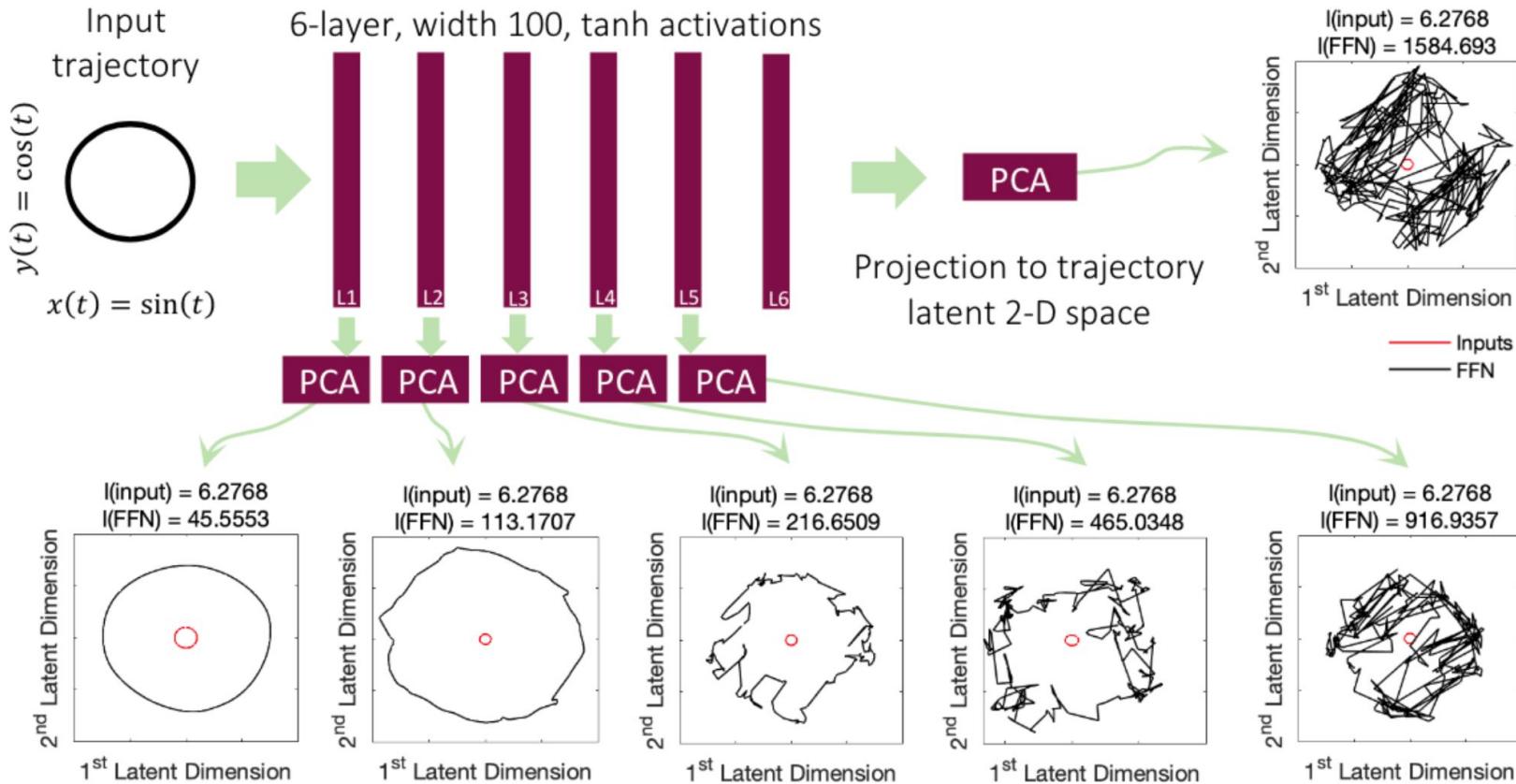
Expressive power

- LTCs are universal approximators (proof in the paper).
 - However, this notion does not help us to *quantify* the expressive power.

Authors resort to the expressivity measure known as trajectory length (Raghu et al. 2017); in brief:

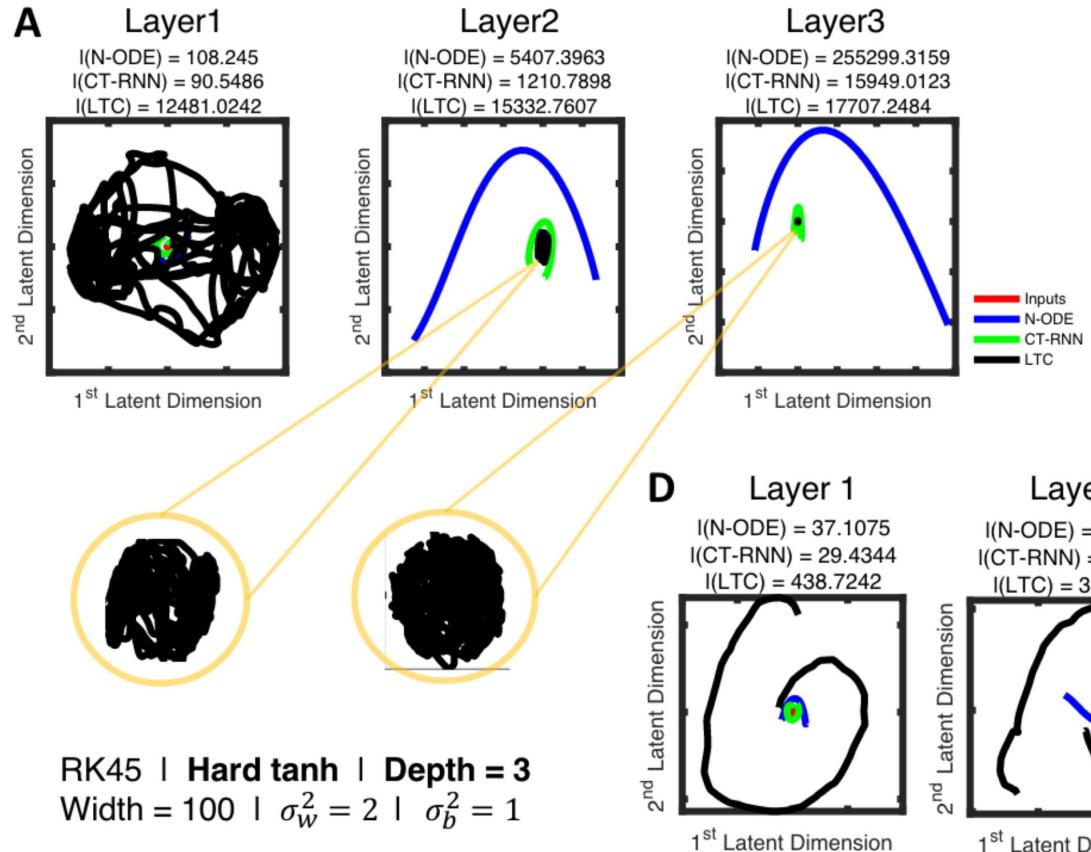
- feed the network with a data sampled from a predefined input trajectory (e.g. 2D circle)
- capture the output of the hidden/latent layers
- pass them through the PCA
- measure the length of the trajectory traversed by the 2 main components in response to the input.

Expressive power: trajectory length of an MLP



Expressive power: trajectory length for LTCs

(contrasted with other types of ODE-based RNNs)



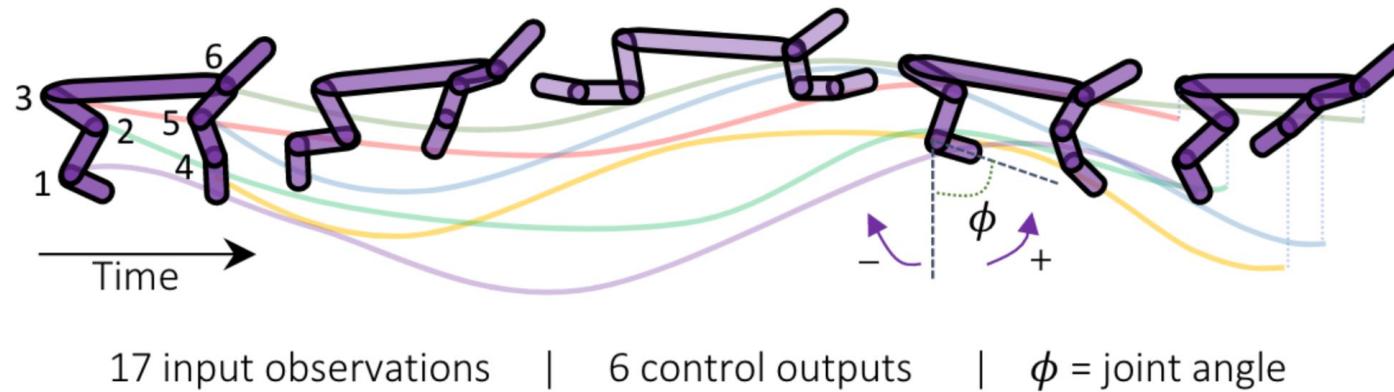
Experimental results: Benchmarks

Notice pretty good performance of LSTMs!

Dataset	Metric	LSTM	CT-RNN	Neural ODE	CT-GRU	LTC (ours)
Gesture	(accuracy)	$64.57\% \pm 0.59$	$59.01\% \pm 1.22$	$46.97\% \pm 3.03$	$68.31\% \pm 1.78$	$69.55\% \pm 1.13$
Occupancy	(accuracy)	$93.18\% \pm 1.66$	$94.54\% \pm 0.54$	$90.15\% \pm 1.71$	$91.44\% \pm 1.67$	$94.63\% \pm 0.17$
Activity recognition	(accuracy)	$95.85\% \pm 0.29$	$95.73\% \pm 0.47$	$97.26\% \pm 0.10$	$96.16\% \pm 0.39$	$95.67\% \pm 0.575$
Sequential MNIST	(accuracy)	$98.41\% \pm 0.12$	$96.73\% \pm 0.19$	$97.61\% \pm 0.14$	$98.27\% \pm 0.14$	$97.57\% \pm 0.18$
Traffic	(squared error)	0.169 ± 0.004	0.224 ± 0.008	1.512 ± 0.179	0.389 ± 0.076	0.099 ± 0.0095
Power	(squared-error)	0.628 ± 0.003	0.742 ± 0.005	1.254 ± 0.149	0.586 ± 0.003	0.642 ± 0.021
Ozone	(F1-score)	0.284 ± 0.025	0.236 ± 0.011	0.168 ± 0.006	0.260 ± 0.024	0.302 ± 0.0155

Experimental results: Physics simulation/modeling

Half-cheetah physics simulation



Algorithm	MSE
LSTM	2.500 ± 0.140
CT-RNN	2.838 ± 0.112
Neural ODE	3.805 ± 0.313
CT-GRU	3.014 ± 0.134
LTC (ours)	2.308 ± 0.015

Module 5: Deep clustering

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2019/2020/2021

<http://www.cs.put.poznan.pl/kkrawiec/>



Outline

1. Introduction
2. Review of deep clustering methods
3. Deep Clustering via Cluster Hardening (DEC)
4. Regularization of clustering

Introduction

Clustering

- Unsupervised assignment of data points to clusters.
- Formally: partitioning of the dataset.
 - Clusters are disjoint.
- Can be based on various underlying concepts:
 - distances, centroids, connectivity, connectivity graphs, density, distribution, ...
- Notable representatives: k-means, k-medoids, agglomerative clustering, Isodata, DBScan, ...
- Implicitly, we often expect a clustering algorithm to discover:
 - the (unknown) decision classes, or
 - a partitioning that generalizes the decision classes, or
 - a partitioning that specializes the decision classes.

Why bother about clustering?

Clustering does away without labels.

- Labelled data is scant and/or hard to come by.
- Labels can be subjective and impose our biases (perceptual, conceptual, ...^{*}) on the system being trained.

The appeal of unsupervised learning.

- On its prevalence:
 - *If intelligence is a cake, the bulk of the cake is unsupervised learning, the icing on the cake is supervised learning, and the cherry on the cake is reinforcement learning.* – Yann LeCun
- Other notable example of unsupervised learning: learning via autoassociation (autoencoders)

*And, as discussed more recently, social, racial, political, ...

'Information richness' for different types of tasks

LeCun's cake metaphor slide, presented at NIPS'16.
Key message: unsupervised learning (especially autoassociation) demands most from a learner, and provides it thus with the richest training signal.

The version presented at ISSCC'19 uses 'self-supervised' instead of unsupervised learning.

<https://www.youtube.com/watch?v=Ount2Y4qxQo&t=1072s>

■ "Pure" Reinforcement Learning (cherry)

- ▶ The machine predicts a scalar reward given once in a while.
- ▶ **A few bits for some samples**



■ Supervised Learning (icing)

- ▶ The machine predicts a category or a few numbers for each input
- ▶ Predicting human-supplied data
- ▶ **10→10,000 bits per sample**

■ Unsupervised/Predictive Learning (cake)

- ▶ The machine predicts any part of its input for any observed part.
- ▶ Predicts future frames in videos
- ▶ **Millions of bits per sample**

■ (Yes, I know, this picture is slightly offensive to RL folks. But I'll make it up)

Example: Efforts on bias mitigation in data labeling

Section “Ground Truth” (5 out of 42 checklist elements):

- *Definition of ground truth reference standard, in sufficient detail to allow replication*
- *Rationale for choosing the reference standard (if alternatives exist)*
- *Source of ground truth annotations; qualifications and preparation of annotators*
- *Annotation tools*
- *Measurement of inter- and intrarater variability; methods to mitigate variability and/or resolve discrepancies*

On the clustering task

Bad news: Clusters are not well-defined.

- The general goal: partition the set of observations into subgroups/clusters of examples that exhibit high degree of similarity within clusters and low degree of similarity between clusters.
- Other desired features of a clustering algorithm:
 - Produce a model applicable to new (test) data.
 - I.e. new observations can be assigned to the learned clusters.
 - Estimate the number of clusters automatically.
 - Meet other expectations, e.g. make the cluster sizes roughly balanced.

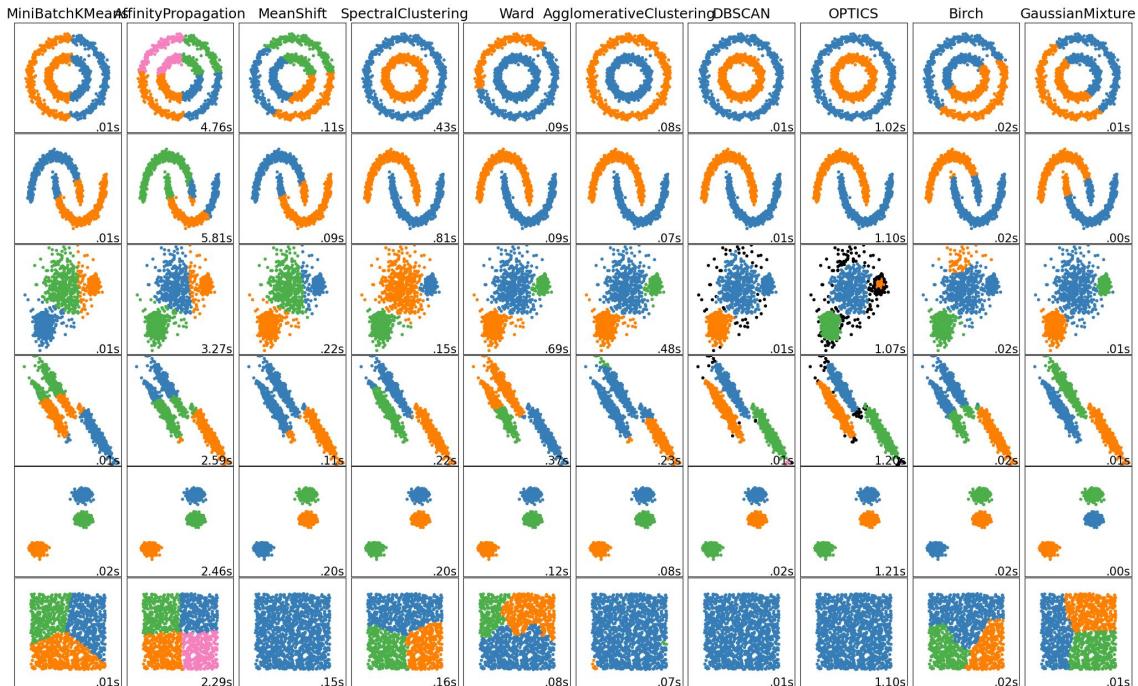
Good news: For some types of models, user's degree of satisfaction with the model can be expressed in a formula.

- That formula can be turned into a loss function that can be optimized with gradient-based methods.

There are clusters and clusters

Various types of clusters and how clustering algorithms fare on them.

The choice of the ‘right’ definition/class of clusters often depends strongly on the nature of the problem.



Review of deep clustering methods

Clustering with Deep Learning: Taxonomy and New Methods

Elie Aljalbout, Vladimir Golkov, Yawar Siddiqui, Maximilian Strobel, Daniel Cremers

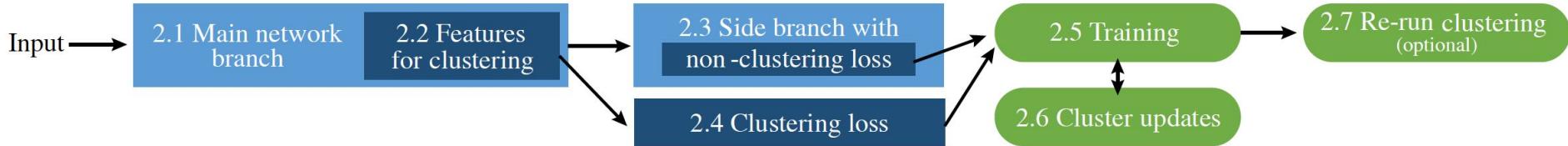
<https://arxiv.org/abs/1801.07648>

A review on deep clustering methods.

Taxonomy

- Neural network training, including:
 - Neural network:
 - Architecture,
 - Set of features used for clustering,
- Loss functions:
 - Non-clustering loss,
 - Clustering loss,
 - Method to combine these two losses,
- Cluster update rule.
- (Optional) Re-running clustering after network training

Paper organization/ reflects, to an extent, the steps of training procedure:



Deep features used for clustering

In DL architectures, there is no need to strictly delineate the feature extraction stage and the ‘proper’ model building stage.

- These co-occur seamlessly along the processing (unless mandated by the architecture, or other aspect)
- Gradient resulting from the loss function is transformed by the clustering network and informs the feature extraction phase.
 - The feature extraction phase becomes thus better informed than in traditional feature selection/construction methods, which typically rely on filter or wrapper approach.

The authors review architectures that ‘harvest’ features:

- From a single (last) layer
- From several layers
 - Note: ‘global’ features can be collected from entire layers via pooling or aggregation
 - Applicable to arbitrary data, not only images.

Neural architectures for clustering

- Multilayer Perceptron
- Convolutional Neural Network
- Deep Belief Network:
 - Generative graphical model, consisting of several layers of latent variables.
- Generative Adversarial Network (GAN):
 - A system of two competing neural network models G and D that engage in a zero-sum game.
- Variational Autoencoder (VAE):
 - A Bayesian network with an autoencoder architecture that learns the data distribution (generative model).

Practical upshot:

- Essentially any DL architecture can be used as a source of features for clustering; it implements an embedding of examples in a feature space.
- Clustering loss can be used as an extra loss term for the ‘core’ task of the network.

Clustering loss: hard assignment

k-Means loss: Assures that the new representation is ‘k-means-friendly’ (Yang et al., 2016a)

$$L(\theta) = \sum_{i=1}^N \sum_{k=1}^K s_{ik} \|z_i - \mu_k\|^2,$$

where:

- z_i : embedded data point,
- μ_k : cluster center (centroid),
- s_{ik} : a binary variable, set to 1 if μ_k is the closest center for z_i ; 0 otherwise

Question 1: How to implement this using typical DL components?

Question 2: What is the main weakness of this loss definition?

Clustering loss: hard assignment

Question 1: How to implement this using typical DL components?

Answer:

- Tensor broadcasting
- Square error
- Minimum (we don't even need to explicitly define the s_{ik})

Question 2: What is the main weakness of this loss definition?

Answer:

- It's discontinuous because of the assignment variable s_{ik}
- Can destabilize the gradients and thus the entire training process
- This can be addressed by resorting to soft assignment of objects to clusters (see next slides).

Clustering loss: soft assignment

Student's t-distribution can be used as the kernel to measure the similarity (van der Maaten and Hinton, 2008) between points z_i and centroids μ_j :

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\nu)^{-\frac{\nu+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2/\nu)^{-\frac{\nu+1}{2}}},$$

where ν is a constant (typically 1).

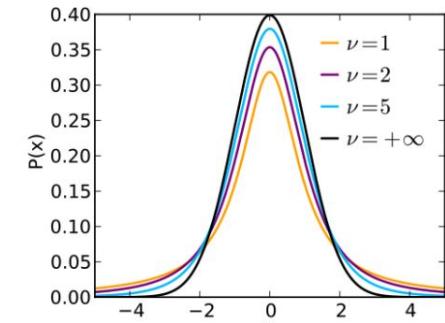
- Compare to the PDF of the t-distribution:
- Note the minus sign in the exponents!

The q_{ij} s:

- are normalized,
- implement soft assignment of objects to clusters,

thus can be used to replace the s_{ik} in the cluster assignment loss.

Question: Can they, really? We don't know what is the desired distribution over clusters.



$$\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Clustering loss: soft assignment

Student's t-distribution can be used as the kernel to measure the similarity (van der Maaten and Hinton, 2008) between points z_i and centroids μ_j :

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\nu)^{-\frac{\nu+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2/\nu)^{-\frac{\nu+1}{2}}},$$

Notice: q_{ij} s define a probability distribution Q . If we had a reference distribution P , we could define our loss as KL divergence:

$$L = \text{KL}(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

Question: Where can we get P from?

Where can we get P from?

Naive option: define P as delta distribution (one-hot vector).

However, because Q is soft, it's better to define P also as a soft distribution.

What do we want?

1. Strengthen predictions (i.e., improve cluster purity),
2. Put more emphasis on data points assigned with high confidence, and
3. Normalize loss contribution

These motivations come from:

Junyuan Xie, Ross Girshick, Ali Farhadi, *Unsupervised Deep Embedding for Clustering Analysis*, <https://arxiv.org/abs/1511.06335> (the next paper presented in this module).

Clustering loss: soft assignment

$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2/\nu)^{-\frac{\nu+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2/\nu)^{-\frac{\nu+1}{2}}},$$

By squaring the original distribution Q and then normalizing it, the auxiliary distribution P forces assignments to have stricter probabilities (closer to 0 or 1).

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_{j'} (q_{ij'}^2 / \sum_i q_{ij'})}.$$

Loss: the KL divergence between these two distributions:

$$L = \text{KL}(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

This is called cluster assignment hardening loss (Xie, Girshick, Farhadi 2016)

Comments

This may be considered a form of self-training (Nigam & Ghani, 2000):

1. We label the original (unlabelled) dataset by applying the initial ‘classifier’ C (clusterer) to it,
2. The labels produced by C (here: after hardening) become targets for C in the successive steps/iterations of training.

The notion of label/distribution hardening is quite often used in many other branches of deep learning.

- In general, a modification of distribution that makes it more ‘extreme’
 - And therefore reduces its entropy.
- Also known as ‘sharpening’.

Deep Clustering via Cluster Hardening

Unsupervised Deep Embedding for Clustering Analysis

Junyuan Xie, Ross Girshick, Ali Farhadi

<https://arxiv.org/abs/1511.06335>

- Method's name: DEC: Deep Embedded Clustering
- One of many methods for Deep Clustering (though a well cited one).

General motivations and the core idea

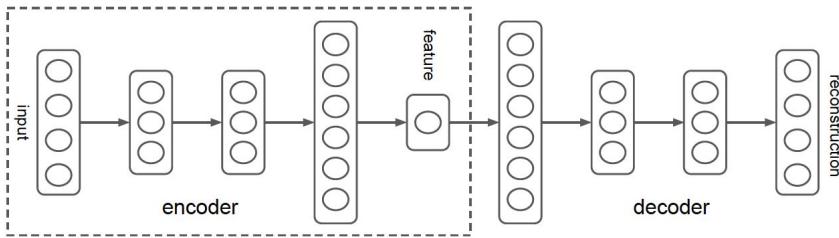
Motivating example: clustering of image data:

- Images carry a lot of internal structure:
 - Correlation of characteristics of neighboring pixels
 - Spatial relationships between objects
 - Random images are highly improbable
- Possible approach: autoencoder
 - Train autoencoder on unlabelled data
 - Use the latent layer as the representation in which the clustering is conducted
- A form of representation learning.

The authors demonstrate this approach on two tasks: clustering of images and clustering of texts.

Training: Phase 1

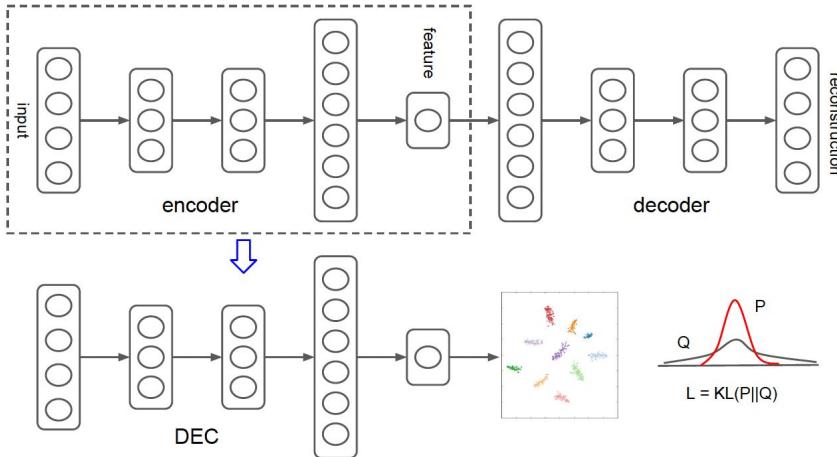
1. Network architecture: “Denoising” stacked autoencoder
2. Iterative training of successive pairs of downstream-upstream layers (from outer to inner) using input-output MSE (also: ReLU, dropout).
 - a. There is thus a series of autoencoders being trained here.
3. Trained layers stacked into a deep autoencoder and fine-tuned (SGD)
4. Resulting encoder (its latent representation) used as feature space



Training: Phase 2 (DEC)

1. Run traditional k-means to obtain the initial locations of centroids μ_k (20 restarts, picking the best solution)
2. Calculate Q, P and KL loss
3. Update the centroids
4. If more than N% of examples changed assignment, go to step 2

Notice: The only parameters: μ_k (ν is a hyperparameter, $\nu=1$)



$$q_{ij} = \frac{(1 + \|z_i - \mu_j\|^2 / \nu)^{-\frac{\nu+1}{2}}}{\sum_{j'}(1 + \|z_i - \mu_{j'}\|^2 / \nu)^{-\frac{\nu+1}{2}}},$$

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_{j'}(q_{ij'}^2 / \sum_i q_{ij'})}.$$

$$L = \text{KL}(P\|Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

The metric: unsupervised clustering accuracy

Quality metric: unsupervised clustering accuracy: finds the optimal matching between the output of a clustering algorithm and a ground-truth assignment of examples to clusters:

$$ACC = \max_m \frac{\sum_{i=1}^n \mathbf{1}\{l_i = m(c_i)\}}{n}$$

where:

- l_i is the ground-truth label,
- c_i is the cluster assignment produced by the algorithm,
- m ranges over all possible one-to-one mappings between clusters and labels,
- n is the number of examples.

Notice:

- This assumes ‘optimistic assignment’: a cluster is labelled according to the majority class of examples that belong to it
- Can be efficiently calculated using the Hungarian method.
 - Linear assignment problem, a.k.a. maximum weight matching in bipartite graph.

The metric: unsupervised clustering accuracy

```
def acc(y_true, y_pred):
    """
    Calculate clustering accuracy. Require scikit-learn installed

    # Arguments
        y: true labels, numpy.array with shape `(n_samples,)`
        y_pred: predicted labels, numpy.array with shape `(n_samples,)`

    # Return
        accuracy, in [0,1]
    """
    y_true = y_true.astype(np.int64)
    assert y_pred.size == y_true.size
    D = max(y_pred.max(), y_true.max()) + 1
    w = np.zeros((D, D), dtype=np.int64)
    for i in range(y_pred.size):
        w[y_pred[i], y_true[i]] += 1
    from sklearn.utils.linear_assignment_ import linear_assignment
    ind = linear_assignment(w.max() - w)
    return sum([w[i, j] for i, j in ind]) * 1.0 / y_pred.size
```

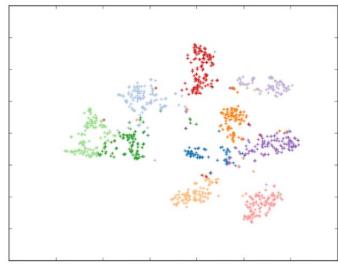
Empirical evaluation

Results:

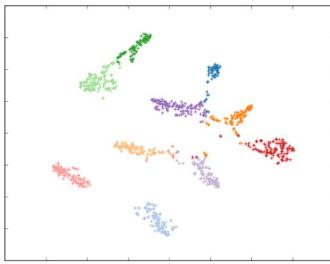
Method	MNIST	STL-HOG	REUTERS-10k	REUTERS
k -means	53.49%	28.39%	52.42%	53.29%
LDMGI	84.09%	33.08%	43.84%	N/A
SEC	80.37%	30.75%	60.08%	N/A
DEC w/o backprop	79.82%	34.06%	70.05%	69.62%
DEC (ours)	84.30%	35.90%	72.17%	75.63%

- Two image datasets, two text datasets (represented as TFIDF of 2000 most frequent word stems)
 - All datasets L2-normalized $\frac{1}{d} \|x_i\|_2^2$
- DEC: the full method
 - encoder receives the gradient from the loss function and is being updated
- DEC w/o backprop:
 - encoder is frozen
- LDMGI and SEC: spectral clustering methods

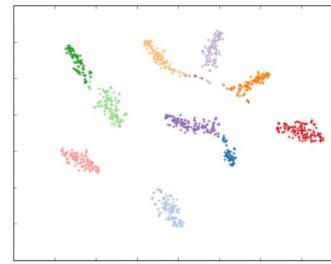
Latent space adaptation during learning (MNIST)



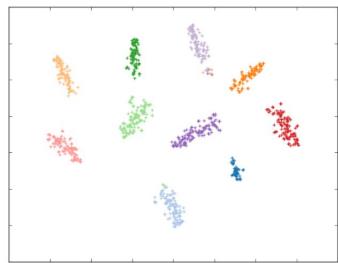
(a) Epoch 0



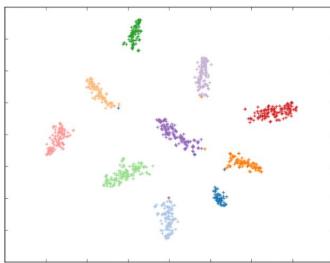
(b) Epoch 3



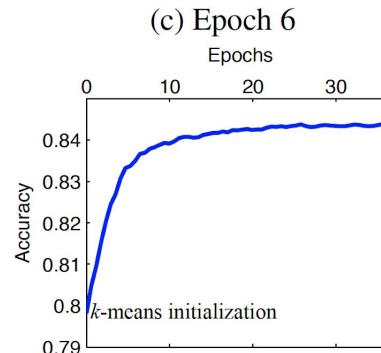
(c) Epoch 6



(d) Epoch 9



(e) Epoch 12

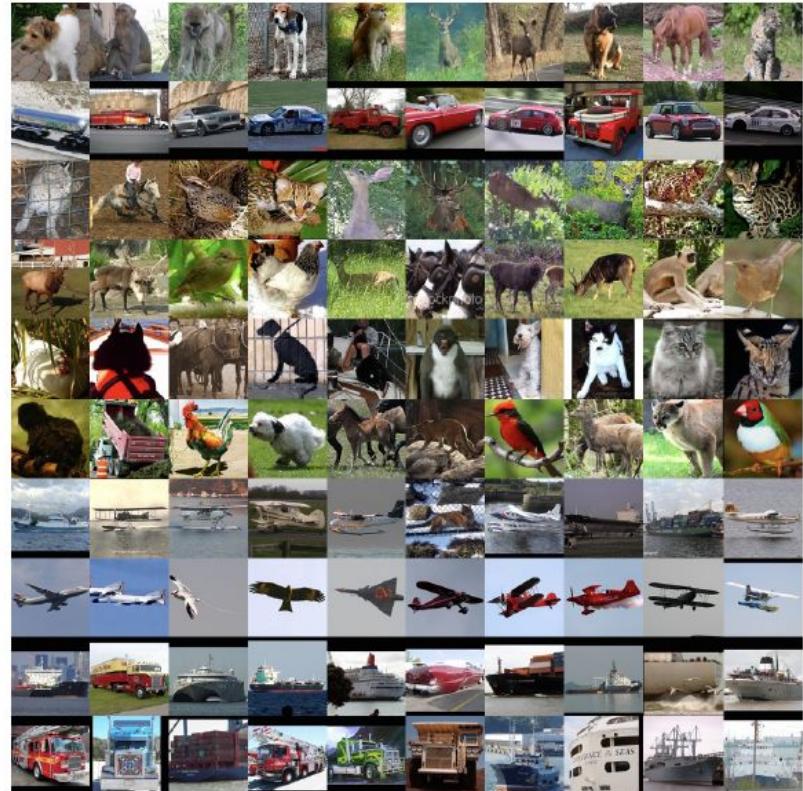


(f) Accuracy vs. epochs

Visualization: t-SNE



(a) MNIST



(b) STL-10

Each row: The 10 top scoring elements from a cluster.

Comment

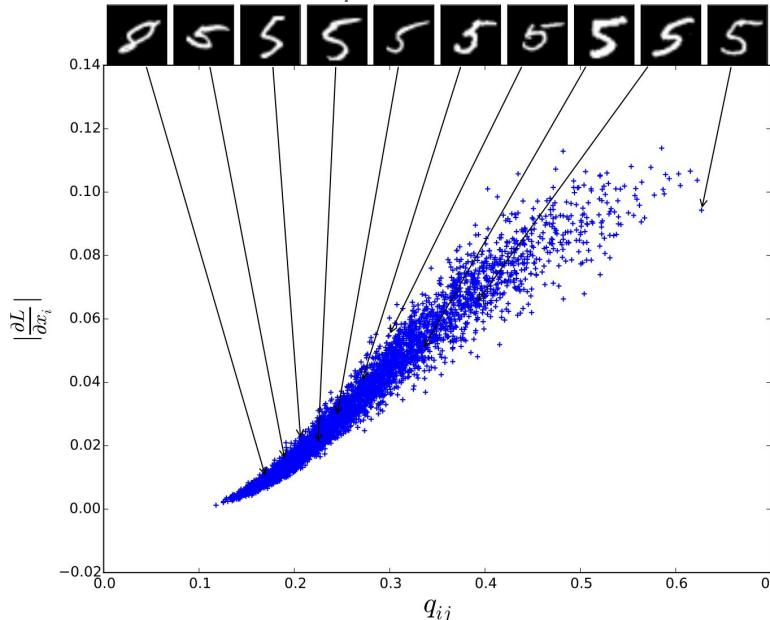
The underlying assumption of DEC is that the initial classifier's high confidence predictions are mostly correct. (Section 5.1)

True for the datasets considered in the study:

- Figure: dependency of gradient on the soft assignment q_{ij} for a random MNIST cluster, at the beginning of optimization.
- Examples closer to the cluster center (higher q_{ij}) contribute more to gradient.

My personal experience:

Much harder to achieve good results on demanding problems, with less compact clusters.



Regularization of clustering: Non-clustering loss components

(continuation of Aljalbout et al.)

Controlling cluster size distribution

Promoting balanced clusters:

$$L_{ba} = \text{KL}(G\|U)$$

U : uniform distribution, G : probability distribution of assigning points to clusters:

$$g_k = P(y = k) = \frac{1}{N} \sum_i q_{ik}$$

Promoting a specific (non-uniform) distribution of cluster sizes:

- Replace U with a distribution of target clusters, based on some premises.
- E.g., Zipf's law: The most frequent word occurs approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.

The rank-frequency distribution is an inverse relation
(k : rank, s : exponent)

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^N (1/n^s)}$$

Locality-preserving loss

Meant to preserve the locality of the mapping from the original space (X, e.g. images) to the latent space (Z). Possible implementation:

$$L_{lp} = \sum_i \sum_{j \in N_k(i)} s(x_i, x_j) \|z_i - z_j\|^2$$

- $N_k(i)$: set of k nearest neighbors of x_i
 - Or, in the simplest case, all xs in the training set.
- $s(x_i, x_j) \in (0,1]$: similarity measure between x_i and x_j
- The products of s and distance in Z captures ‘decorrelation’.

Note:

- Does not explicitly refer to cluster centers.
- Measuring similarity in X is in general challenging.

Other non-clustering loss terms used in clustering

- Autoencoder reconstruction loss:

$$L = d_{\text{AE}}(x_i, f(x_i)) = \sum_i \|x_i - f(x_i)\|^2,$$

(see next slide)

- Self-Augmentation Loss: Hu et al. (2017): pushes together the representation of the original sample and augmentations of the examples from that sample:

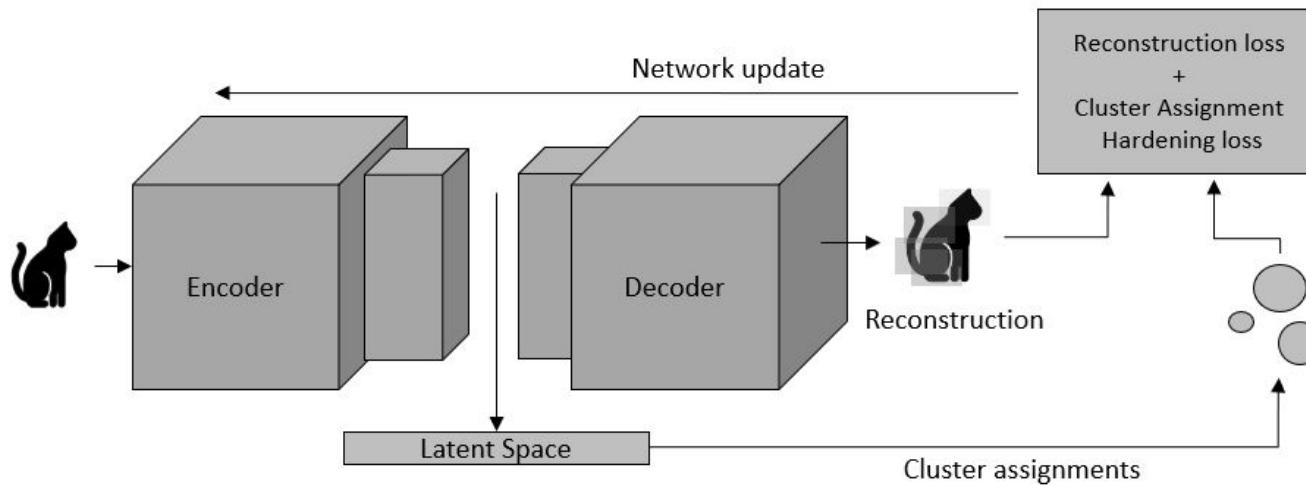
$$L = -\frac{1}{N} \sum_N s(f(x), f(T(x))),$$

where T is an augmentation operation, applied N times (randomized).

Engaging the reconstruction loss

Aljalbout et al.:

- Don't discard the decoder: use the reconstruction loss too.
- Motivation: reconstruction loss forces the latent representation to maintain its high information content.
 - Lower risk of features degeneration under the pressure of clustering loss.



Involving AE reconstruction loss term

Results for
MNIST
(top)
and
COIL-20
(bottom)

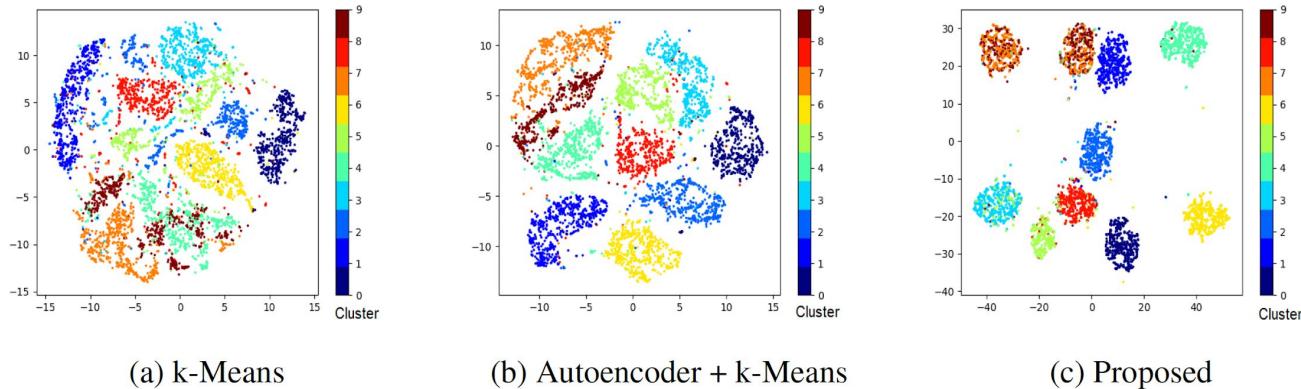
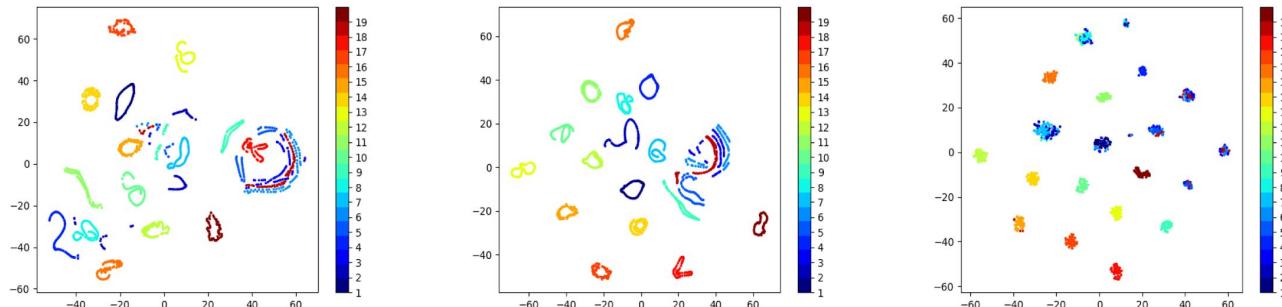


Figure 3: t -SNE visualizations for clustering on MNIST dataset in (a) Original pixel space, (b) Autoencoder hidden layer space and (c) Autoencoder hidden layer space with the proposed method.



Combining loss terms

In general:

$$L(\theta) = \alpha L_c(\theta) + (1 - \alpha)L_n(\theta),$$

L_c : clustering loss, L_n : non-clustering loss.

Ways of determining and scheduling the changes of α :

- Pre-training and fine-tuning: start with $\alpha=0$, then set $\alpha=1$
- Joint training, e.g. $\alpha=0.5$
- Variable schedule: e.g., start with low α and increase it gradually.

Concluding remarks

- Deep clustering allows end-to-end learning of clustering together with representation learning.
 - Joint learning of representation and (soft) cluster assignment.
 - For arbitrary type/domain of the training examples (images, text, ...)
- Deep clustering can be easily hybridized with other types of modules.
 - E.g. autoencoders.
- Loss functions can be easily extended with additional terms that express the desirable properties of the clustering.

Module 6: Graph Neural Networks

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2023-2024

<http://www.cs.put.poznan.pl/kkrawiec/>



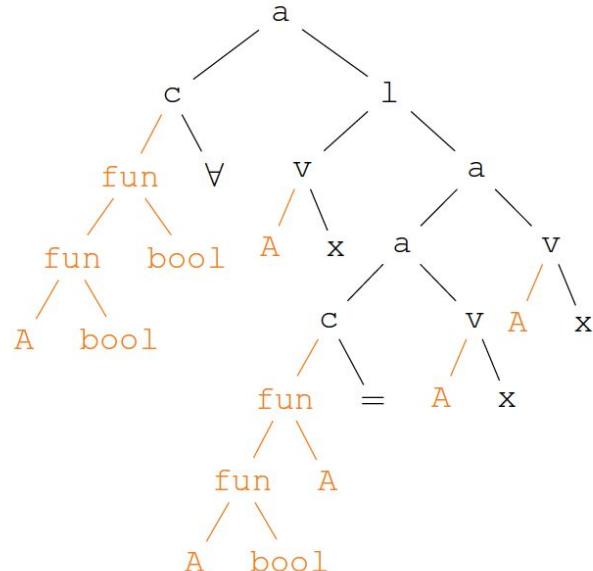
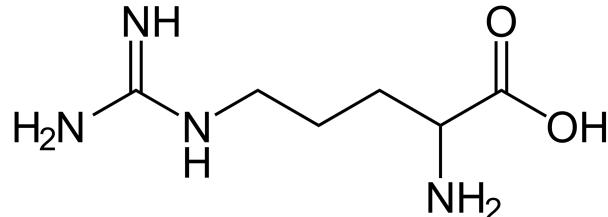
Outline

1. [Introduction](#)
2. [Message-passing GNNs: A use case](#)
3. [Covariant graph networks](#)
4. [Attempts of unification](#)
5. [Summary](#)

Introduction

Motivations for processing graphs

- A range of application areas involve non-tensor data: computer vision, chemistry, network science, programs, ...
- Graphs are natural for modeling many natural phenomena: relationships, flows, dependencies, ...



Graph definition

$$G = (V, E, l_V, l_E)$$

where:

- V is a set of nodes,
- $E \subseteq V \times V$ is a set of [directed] edges,
- $l_V: V \rightarrow L_V$ defines vertex labels,
- $l_E: E \rightarrow L_E$ defines edge labels.

Challenges in processing graphs with NNs

The primary challenge: variable-size structure.

Compared to sequence learning (e.g. using recurrent NNs):

- Sequences are 1D structures; graphs and trees aren't.
- More arbitrariness in traversing the structure:
 - Which node to start with? Which to select next? Or should we parse all nodes in parallel?
- No ‘natural ordering’ of constituents of the structure.
 - More precisely: no *complete* order (*partial* order present for directed graphs).

Therefore:

- Most approaches do not process graph nodes in any specific order.
- Rather than that, they process iteratively the transient states generated for and associated with individual nodes and edges (~ message passing).

Main types of Graph NNs (GNNs)

- Graph convolution
 - Simple local aggregation of node embeddings over local neighborhoods, for each node independently
- Message-passing
 - A separate submodel calculates the effects of interaction of a given node with each neighbor node independently.
 - Aggregation of messages over the neighborhood.
- Graph-2-graph transducers
- Generic models (e.g. Battaglia et al., see the “Unification” section later in this module)

Message-passing GNNs: A use case

Graph Representations for Higher-Order Logic and Theorem Proving

Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, Christian Szegedy

<https://arxiv.org/abs/1905.10006>

Application area

- The goal: To improve the effectiveness of theorem provers.
- Key question/challenge: How to efficiently navigate in the vast space of premises (and corresponding paths of proof) during theorem proving?

The benchmark:

- HOList (Bansal et al. 2019): stateless, higher-order theorem proving API
<https://sites.google.com/view/holist/home>
- More than 20k mathematical theorems and their proofs (human-constructed).
- The HOList proof assistant allows ML algorithms to interact with the HOL Light interactive theorem prover.

Examples

```
let fib = define
  `fib n = if n = 0 \/\ n = 1 then 1 else fib(n - 1) + fib(n - 2)`;;

let fib2 = define
  `(fib2 0 = 1) /\ 
   (fib2 1 = 1) /\ 
   (fib2 (n + 2) = fib2(n) + fib2(n + 1))`;; 

let halve = define `halve (2 * n) = n`;; 

let unknown = define `unknown n = unknown(n + 1)`;; 

define
  `!n. collatz(n) = if n <= 1 then n
                      else if EVEN(n) then collatz(n DIV 2)
                      else collatz(3 * n + 1)`;;
```

ARITH_RULE

```
`(a * x + b * y + a * y) EXP 3 + (b * x) EXP 3 +
 (a * x + b * y + b * x) EXP 3 + (a * y) EXP 3 =
 (a * x + a * y + b * x) EXP 3 + (b * y) EXP 3 +
 (a * y + b * y + b * x) EXP 3 + (a * x) EXP 3`;;
```

Data representation

S-expressions, e.g.: $f(x)$ is represented as

$$(a (v (\text{fun } A B) f) (v A x)))$$

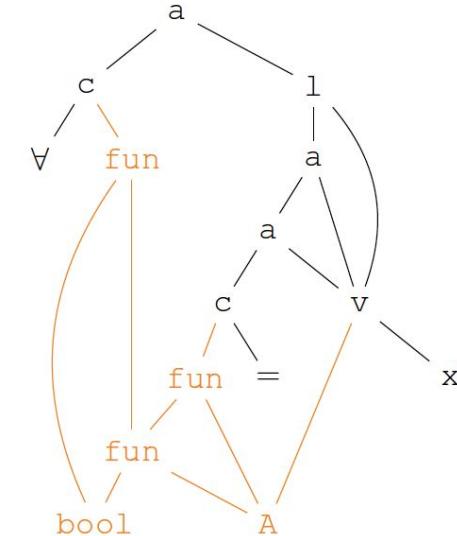
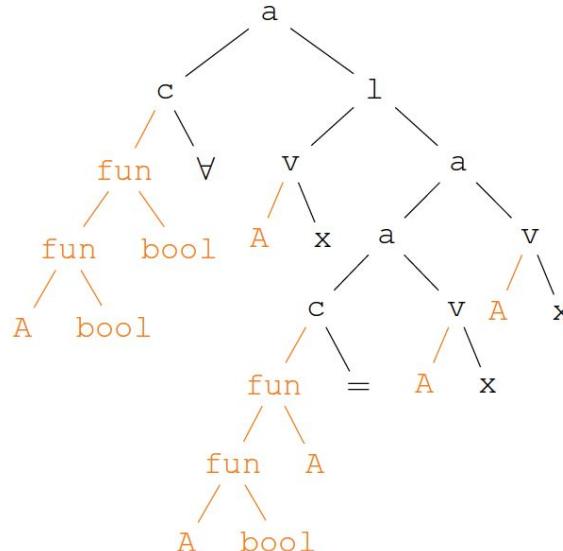
- a : function application
- v : is-a-variable; $(v A a)$ states “ a is a variable of type A ”
- A, B : input and output types
- $(\text{fun } A B)$: defines the type of functions from A to B

The authors consider also variants with bottom-up edges (each child points to its parents).

Abstract Syntax Trees: Tree vs. graph

Representation of the axiom: $\forall x : x = x$.

- Left: as a tree.
- Right: as a graph.



Graph representations are more compact by avoiding subtree duplication (which requires potentially costly testing for equivalence).

- It is easier to find canonical forms for graphs.

Proof search

Iterative process, in which each step decisions have to be made concerning:

1. Which of (predefined) tactics (predefined proof steps/techniques) to apply to the current search state?
2. Which of preceding theorems (premises) to pick as the basis for this current proof step (goal)?

These tasks are normally conducted by humans in proof assistants (see e.g. Isabelle, Coq). To automate them, we need two components:

1. Tactic scorer
2. Premise scorer

The authors implement them as deep neural networks, with message-passing GNNs as components to ‘parse’ graphs that represent goals and premises.

Graph definition

$$G = (V, E, l_V, l_E)$$

where:

- V is a set of nodes,
- E is a set of directed edges,
- l_V maps nodes to a fixed vocabulary of tokens (*labels*),
- l_E maps edge to a 0/1 encoding the direction of the edge

Definition of l_E can be extended for graphs with labels on edges.

Core component: Message-passing GNN

Graph neural networks (GNNs) compute embeddings for nodes in a graph via consecutive rounds (also called hops) of end-to-end differentiable message passing (MP).

- Iterative aggregation of information ‘received’ by each node from its neighbors, in two phases:
 - Message formation (Step 2 in the slides that follow)
 - Message aggregation and node state update (Step 3)
- The number of iterations (hops) T determines the effective size of the neighborhood (‘receptive field’).
- Each mapping realized with multi-layer perceptrons (MLP).
- The resulting graph has exactly the same topology as the input one.
 - Extra steps required to produce output for classification, regression, etc.

Next slides describe the steps of a single iteration of message passing.

MP: Step 1: Initialization

This step takes place only once per example (graph), at ‘time’ t=1.

Separate embedding of each node v and edge e :

$$\mathbf{h}_v^1 = \text{MLP}_V(\mathbf{x}_{l_V(v)})$$

$$\mathbf{h}_e = \text{MLP}_E(l_E(e))$$

where:

- \mathbf{x}_i : trainable feature vector associated with i-th token (token/label embedding)
- l_E : binary variable indicating the direction of the edge
- MLP: multilayer perceptron – a stack of few nonlinear layers.
 - One could consider using the \mathbf{x}_i embedding directly as \mathbf{h}_v , without the intermediate step of MLP_V .

The authors used $|\mathbf{h}_v|=128$

MP: Step 2: Message formation (per edge)

Generation of messages: for each edge (u, v) , we aggregate

- the embeddings for u , v ,
- and the edge that connects them,

separately for the parent (incoming) nodes (first formula) and the child (outgoing) nodes (second formula):

$$s_{u,v}^t = \text{MLP}_{\text{edge}}^t([h_u^{t-1}, h_v^{t-1}, h_e])$$

$$\hat{s}_{u,v}^t = \hat{\text{MLP}}_{\text{edge}}^t([h_u^{t-1}, h_v^{t-1}, h_e])$$

Note:

- $[, ,]$ denotes vector concatenation.
- Edges have transient ‘states’ associated with them (‘message’).
- The above models keep referring to edge labels h_e throughout all iterations.

MP: Step 3: Message aggregation and state update

Aggregation of messages for each node separately, based on averaged messages:

$$\mathbf{h}_v^t = \mathbf{h}_v^{t-1} + \text{MLP}_{\text{aggr}} \left(\left[\mathbf{h}_v^{t-1}, \sum \frac{\mathbf{s}_{u,v}^t}{p(v)}, \sum \frac{\hat{\mathbf{s}}_{u,v}^t}{c(v)} \right] \right)$$

where:

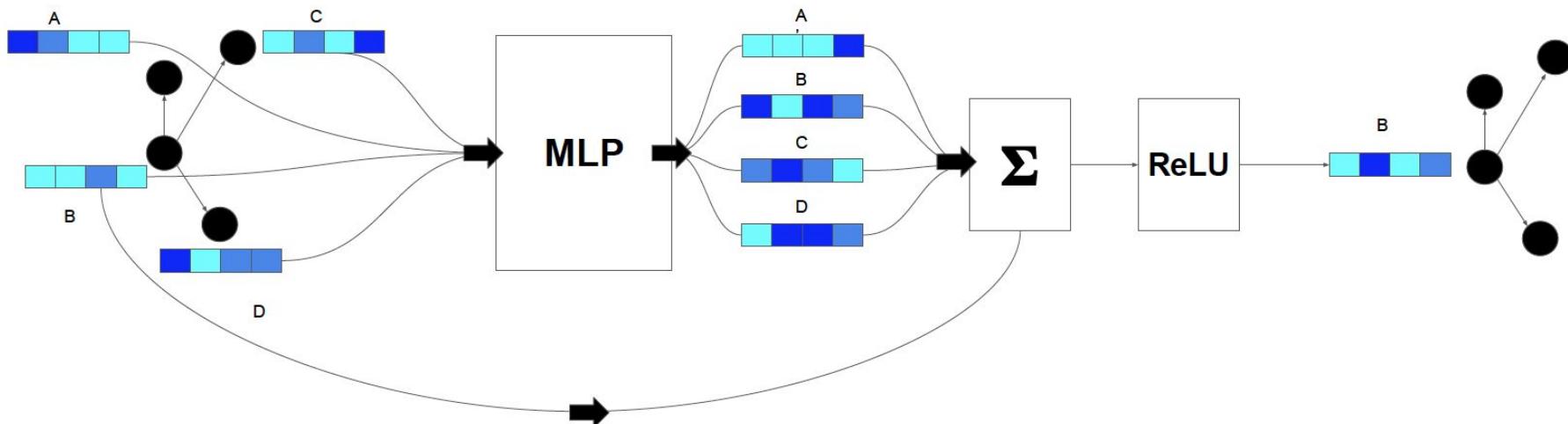
- $p(v)$: the number of parents of v
- $c(v)$: the number of children of v

Comments:

- Notice the incremental character of the formula.
 - Also, \mathbf{h}_v is *both* fed into the aggregating model *and* directly incremented.

Message passing

Single (t -th) step of processing for node B and its neighborhood (warning: this diagram is *simplified*, blurring the distinction between MLPs):



- MLPs applied independently, in parallel, to each node and edge.
- Q: What does the lowermost arrow remind you of?

Message passing: Remarks

- The whole model comprises just five MLPs in total.
 - The formalism becomes even simpler for undirected graphs.
- However, the ‘unrolling’ of the model with time t (the steps of message passing) may cause the *effective depth* of the model to be relatively large
 - and negatively affect propagation of gradient.
- Practical upshot: the number of iterations T is typically moderate.

Comments

Model's perception of the actual graph structure is quite limited. Recall the averaging in the aggregation step:

$$\mathbf{h}_v^t = \mathbf{h}_v^{t-1} + \text{MLP}_{\text{aggr}} \left(\left[\mathbf{h}_v^{t-1}, \sum \frac{\mathbf{s}_{u,v}^t}{p(v)}, \sum \frac{\hat{\mathbf{s}}_{u,v}^t}{c(v)} \right] \right)$$

Limitations:

- The aggregation network has no access to the number of adjacent nodes.
 - Easy to fix/extend: Feed the number of neighbors into an additional input of MLP_{aggr} .
- Different sets of messages can be mapped to the same output.

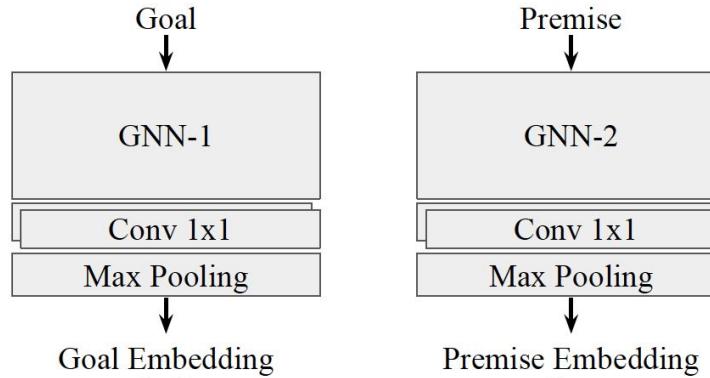
Questions:

- What type of architecture is this message passing GNN?
- How to set the number of steps T? What is its relationship with graph size?

Application: Stage 1: Graph embedding

The approach uses two message-passing GNNs (same architecture, different weights) to independently embed goal graphs and premise graphs.

- Not to be confused with node and edge embeddings used by GNNs internally!

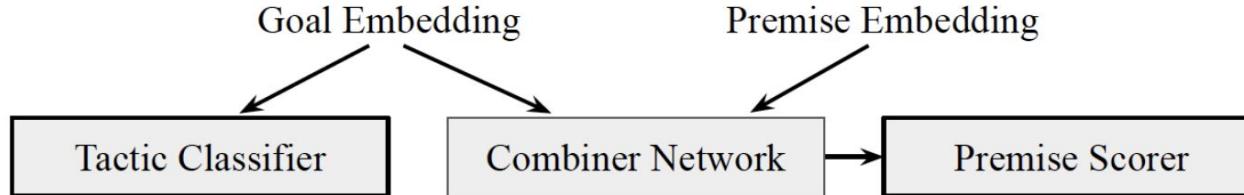


- Dimensionality of node embedding in both GNNs is relatively small (128),
 - so the authors expand them afterwards to 1024 using 1x1 convolutions (dense layer applied to each node independently).
- This is followed by max pooling to aggregate over all graph nodes.

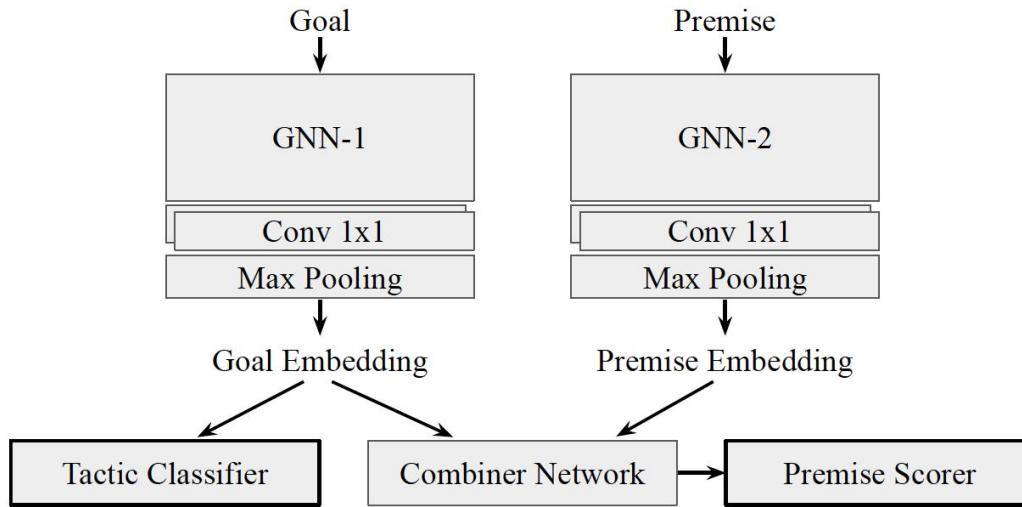
Application: Stage 2: Scoring of tactics and goals

The obtained embeddings of goal and premise graphs are then used to:

1. Score the 41 predefined tactics to be used by the prover (Tactic Classifier, TC)
 - o The input to TC is only the current goal (an intermediate state in proof search)
2. Score the premises to follow in the next step of proof (Premise Scorer, PS)
 - o The input to PS takes into account both the current goal and a candidate premise.



Application: Overall architecture



- Interesting Combiner Network: concatenates both inputs and their elementwise multiplication, and then applies three dense layers.
- GNN-2 applied to premises (preceding theorems) off-line, only once (theorems are given),
 - so only the combiner network has to be applied to all goal-premise combinations for a given (sub)goal

Comments

- Learning mode: Imitation learning: the system learns to mimic the behavior of a human prover.
 - Notice the difference with supervised learning (?)
 - A plausible alternative: reinforcement learning
- Training set derived from 10,200 top-level theorems.
 - In total ~375,000 proof steps (subgoals)
- Test set: 3,225 theorems.
- Each positive example is a triple fetched from a human-constructed proof:
 - the (sub)goal to be proven (at that step),
 - the tactic applied to it,
 - a list of theorem parameters passed to the tactic (or special token for no parameter).
- Negative examples: random pairing of goals with theorems, for theorems that have been used at least once in the training set.

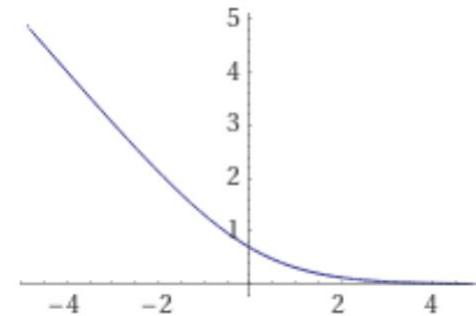
Loss function

Weighted sum of:

- cross-entropy loss of the tactic classifier,
- cross-entropy loss of pairwise premise scorer, and
- AUCROC loss (Burges et al. 2005; Eban et al. 2017):

$$AUCROC_b = \sum_i \sum_j loss(\text{logit}_i - \text{logit}_j)$$

$$loss(l) = \ln(1 + e^{-l})$$



where i ranges over the positive premises in batch b , and j ranges over the negatives in b .

Relation to ranking: an attempt to maximize the spread between the positives and negatives.

Scope of experiments

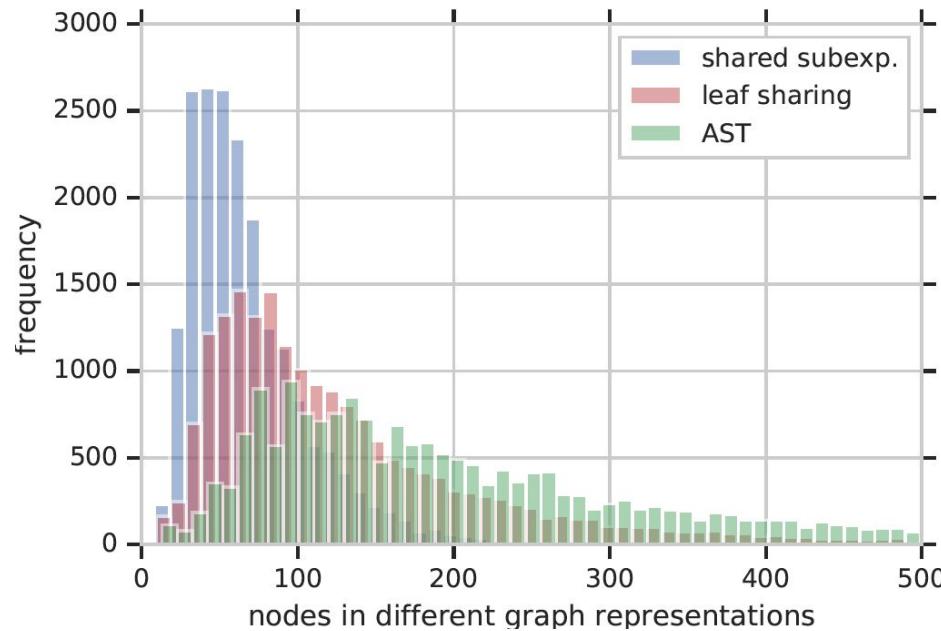
Input graphs are transformed before feeding into a GNN, to reduce their size and ‘canonicalize’ them.

Transformations considered by the authors:

- Sharing nodes of subexpressions (*subexpression sharing*).
- Sharing equal leaves of the AST (*leaf sharing*).
- Replacing all variable names by x (*variable blinding*).
- Removing all edges from nodes to their parents and only keeping those to their children (*top down*).
- Removing all edges from nodes to their children and only keeping those to their parents (*bottom up*).
- Adding random edges (*random*).

Impact of subexpression and leaf sharing

Statistics of the HOList database:



Some results

The metric: Percentage of proofs *closed* on the validation set: running the prover once over the validation set using the parameterized tactics predicted by the system to guide proof search.

Table 1: Our best model, compared against previous state of the art and bag of words model as baselines.

Network Architecture	% Proofs Closed (Validation Set)
Baseline: S-expression as a string	
WaveNet (Bansal et al. 2019)	32.65%
Baseline: Bag of words	
Max pooling only	37.98%
Subexpression sharing	
12-hop GNN	49.95%

Representation	0 Hops	2 Hops	4 Hops	8 Hops	12 Hops
Abstract syntax trees (AST)	40.18%	43.84%	44.58%	46.66%*	45.67%*
Leaf sharing	41.76%	33.89%	29.24%	29.51%	30.51%
Leaf sharing + variable blinding	31.78%	32.18%	32.80%	30.04%	31.00%
Subexpression sharing	40.86%	42.94%	46.94%	47.22%	49.95%
Subexpression sharing + variable blinding	31.75%	34.44%	35.96%	34.07%	37.36%
Subexpression sharing + random	41.24%	43.68%	43.84%	42.63%	42.94%
Subexpression sharing + top down	40.55%	43.59%	45.51%	48.24%	48.40%
Subexpression sharing + bottom up	39.72%	40.58%	41.16%	41.86%	40.99%

Conclusions: More hops helps. Subexpression sharing is essential.

Covariant graph networks

Predicting molecular properties with covariant compositional networks

Truong Son Hy, Shubhendu Trivedi, Horace Pan, Brandon M. Anderson, and Risi Kondor

<https://doi.org/10.1063/1.5024797>

Covariant Compositional Networks For Learning Graphs

Risi Kondor, Hy Truong Son, Horace Pan, Brandon Anderson, Shubhendu Trivedi
<http://arxiv.org/abs/1801.02144>

Summary: Address the question: which types of invariances should we care about when processing graphs?

Motivations

The two major issues that graph learning methods need to grapple with are:

- invariance to permutations,
 - (there is no ordering of adjacent nodes)
- capturing structures at multiple different scales.

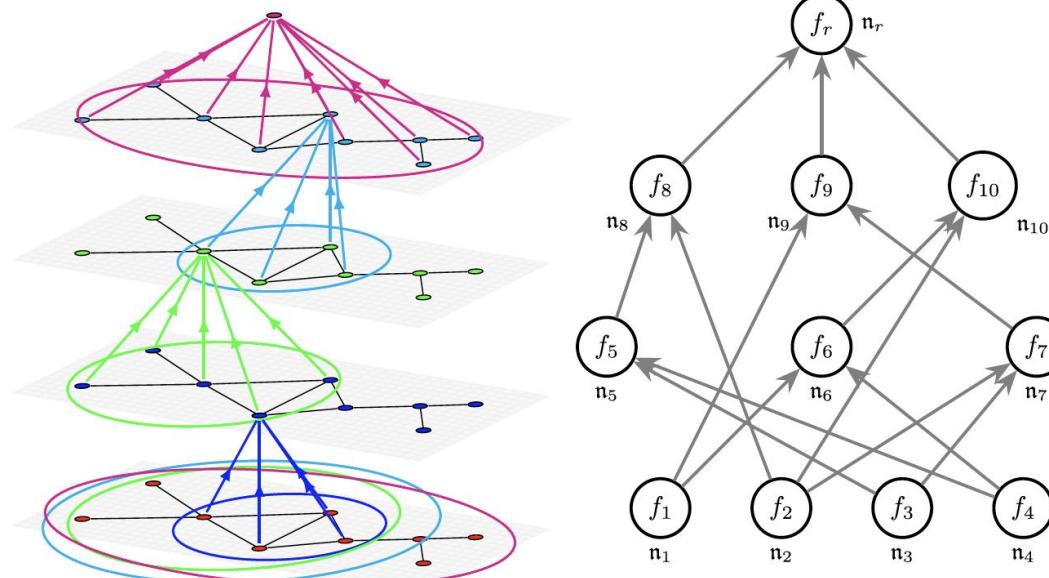
Claim: *Invariant aggregation functions, of the type popularized by message passing neural networks, are not the most general way to build compositional models for compound objects, such as graphs.*

For instance:

- In CNNs, if the input image is translated, the activations in each layer translate in the same way.
- This quasi-invariance is called *equivariance*.

Receptive fields

Left: The multi-layered processing with receptive fields (only some fields shown for clarity). Note the effective receptive fields and the last aggregation step.
Right: the corresponding neural network (simplified for clarity too).



Quasi-invariance in Comp-nets

Quasi-invariance amounts to asserting that the activation f_i at any given node i must only depend on $P_i = (e_{j_1}, \dots, e_{j_k})$ as a set and *not on the internal ordering of the atoms* e_{j_1}, \dots, e_{j_k} making up the receptive field.

However, this property is potentially problematic since we lose all information about which vertex (in the receptive field) has contributed what to the aggregate information (activation).

The solution is to regard the P_i receptive fields as ordered sets and explicitly establish how f_i co-varies with the internal ordering of the receptive fields.

Formal definition of covariance

Definition 2. Assume that \mathcal{N} is the comp-net of a graph \mathcal{G} and \mathcal{N}' is the comp-net of the same graph but after its vertices have been permuted by some permutation σ . Given any $n_i \in \mathcal{N}$ with receptive field $\mathcal{P}_i = (e_{p_1}, \dots, e_{p_m})$, let n'_j be the corresponding node in \mathcal{N}' with receptive field $\mathcal{P}'_j = (e_{q_1}, \dots, e_{q_m})$. Assume that $\pi \in \mathbb{S}_m$ is the permutation that aligns the orderings of the two receptive fields, i.e., for which $e_{q_{\pi(a)}} = e_{p_a}$. We say that the comp-nets are **covariant to permutations** if for any π , there is a corresponding function R_π such that $f'_j = R_\pi(f_i)$.

Implementation

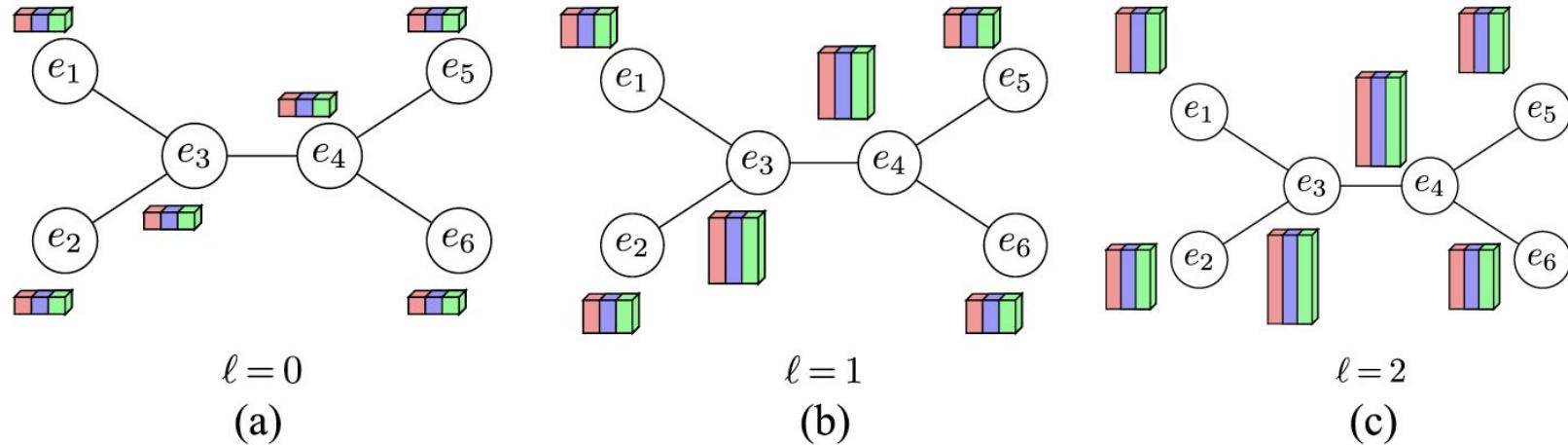
Uses k-order tensors and permutation matrices to implement the covariance.

$$[P_\pi]_{i,j} = \begin{cases} 1 & \text{if } \pi(j) = i, \\ 0 & \text{otherwise.} \end{cases}$$

Represents excitations as vectors and ‘stacks’ permutation matrices using the Kronecker product:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 2 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \\ 3 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} & 4 \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} \end{bmatrix}$$

Visualization of tensors



- The tensors processed in consecutive layers are *covariant* with respect to permutation of adjacent nodes in node's neighborhood,
 - in contrast to conventional message passing, where it is *invariant*.

Benchmarks

Classification of chemical compounds:

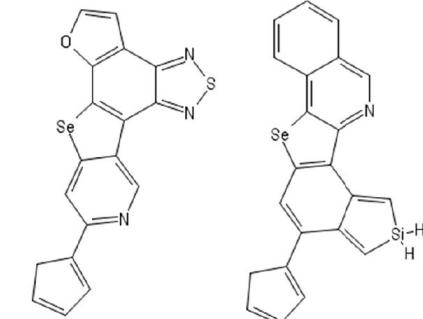
1. MUTAG: 188 mutagenic aromatic and heteroaromatic compounds
2. PTC: 344 chemical compounds that have been tested for positive or negative toxicity in lab rats
3. NCI1 and NCI109: 4110 and 4127 compounds, respectively, each screened for activity against small cell lung cancer and ovarian cancer lines.

Regression of properties of chemical compounds:

- Harvard Clean Energy Project (HCEP): 2.3M organic candidate compounds for use in solar cells
 - Inputs: molecular graphs (derived from their SMILES strings),
 - Regression target: power conversion efficiency (PCE).

Other: QM9: 134K organic molecules.

- Target: 13 molecular properties (multivariate regression)



Some results

TABLE I. Classification of results on the kernel datasets (accuracy \pm standard deviation).

	MUTAG	PTC	NCI1	NCI109
Wesifeiler–Lehm̄an kernel ³³	84.50 ± 2.16	59.97 ± 1.60	84.76 ± 0.32	85.12 ± 0.29
Wesifeiler–Lehman edge kernel ³³	82.94 ± 2.33	60.18 ± 2.19	84.65 ± 0.25	85.32 ± 0.34
Shortest path kernel ²⁹	85.50 ± 2.50	59.53 ± 1.71	73.61 ± 0.36	73.23 ± 0.26
Graphlets kernel ³¹	82.44 ± 1.29	55.88 ± 0.31	62.40 ± 0.27	62.35 ± 0.28
Random walk kernel ²⁸	80.33 ± 1.35	59.85 ± 0.95	Timed out	Timed out
Multiscale Laplacian graph kernel ⁶²	87.94 ± 1.61	63.26 ± 1.48	81.75 ± 0.24	81.31 ± 0.22
PSCN($k = 10$) ³⁷	88.95 ± 4.37	62.29 ± 5.68	76.34 ± 1.68	N/A
Neural graph fingerprints ²¹	89.00 ± 7.00	57.85 ± 3.36	62.21 ± 4.72	56.11 ± 4.31
Second order CCN (our method)	91.64 ± 7.24	70.62 ± 7.04	76.27 ± 4.13	75.54 ± 3.36

Results for the regression task

TABLE II. HCEP regression results. Error of predicting power conversion efficiency in units of percent. (Best results indicated in bold.)

	Test MAE	Test RMSE
Lasso	0.867	1.437
Ridge regression	0.854	1.376
Random forest	1.004	1.799
Gradient boosted trees	0.704	1.005
Weisfeiler–Lehman kernel ³³	0.805	1.096
Neural graph fingerprints ²¹	0.851	1.177
PSCN ($k = 10$) ³⁷	0.718	0.973
Second order CCN (our method)	0.340	0.449

QM9 (multivariate regression)

Properties:

- (a) U_0 : atomization energy at 0 K (eV),
- (b) U : atomization at room temperature (eV),
- (c) H : enthalpy of atomization at room temperature (eV),
- (d) G : free energy of atomization (eV),
- (e) ω_1 : highest fundamental vibrational frequency (cm^{-1}),
- (f) ZPVE: zero point vibrational energy (eV),
- (g) HOMO: highest occupied molecular orbital, energy of the highest occupied electronic state (eV),
- (h) LUMO: lowest unoccupied molecular orbital, energy of the lowest unoccupied electronic state (eV),
- (i) GAP: difference between HOMO and LUMO (eV),
- (j) R^2 : electronic spatial extent (bohr^2),
- (k) μ : norm of the dipole moment (D),
- (l) α : norm of the static polarizability (bohr^3),
- (m) C_v : heat capacity at room temperature (cal/mol/K).

TABLE III. QM9(a) regression results (mean absolute error). Here we have only used the graph as the learning input without any physical features. (Best results indicated in bold.)

	WLGK	NGF	PSCN	CCN 2D
α (bohr^3)	3.75	3.51	1.63	1.30
C_v [cal/(mol K)]	2.39	1.91	1.09	0.93
G (eV)	4.84	4.36	3.13	2.75
GAP (eV)	0.92	0.86	0.77	0.69
H (eV)	5.45	4.92	3.56	3.14
HOMO (eV)	0.38	0.34	0.30	0.23
LUMO (eV)	0.89	0.82	0.75	0.67
μ (D)	1.03	0.94	0.81	0.72
ω_1 (cm^{-1})	192.16	168.14	152.13	120.10
R_2 (bohr^2)	154.25	137.43	61.70	53.28
U (eV)	5.41	4.89	3.54	3.02
U_0 (eV)	5.36	4.85	3.50	2.99
ZPVE (eV)	0.51	0.45	0.38	0.35

An attempt of unification

Relational inductive biases, deep learning, and graph networks

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, Razvan Pascanu

<https://arxiv.org/abs/1806.01261>

Motivation

A hybrid of review and position paper. Proposes a general conceptualization.

Questions addressed by the authors:

1. Can we define an overarching formalism for Graph NNs?
2. What are the types of invariance and relational inductive biases embodied by various NN architectures?

Component	Entities	Relations	Rel. inductive bias	Invariance
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Nodes	Edges	Arbitrary	Node, edge permutations

Types of weight sharing in NNs and DL

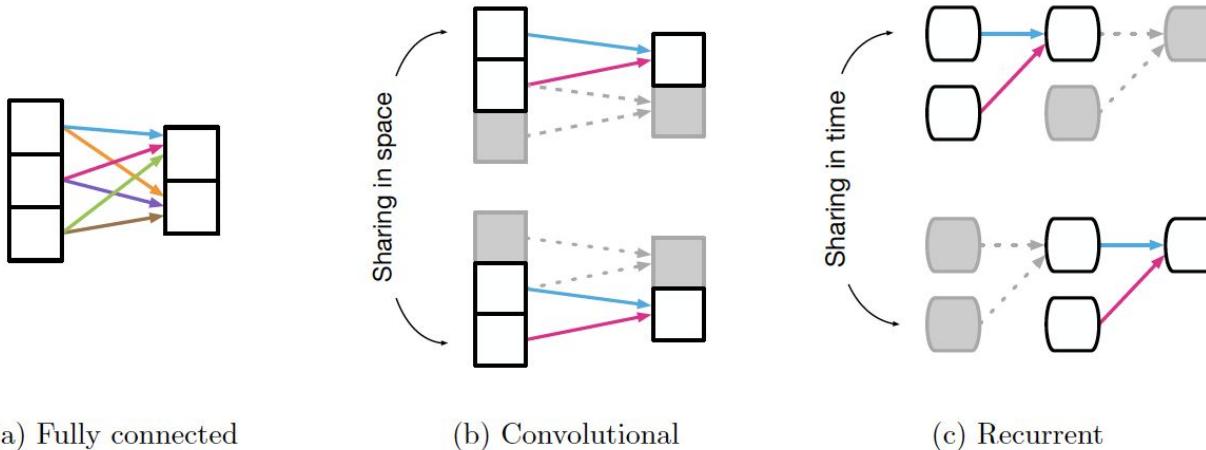
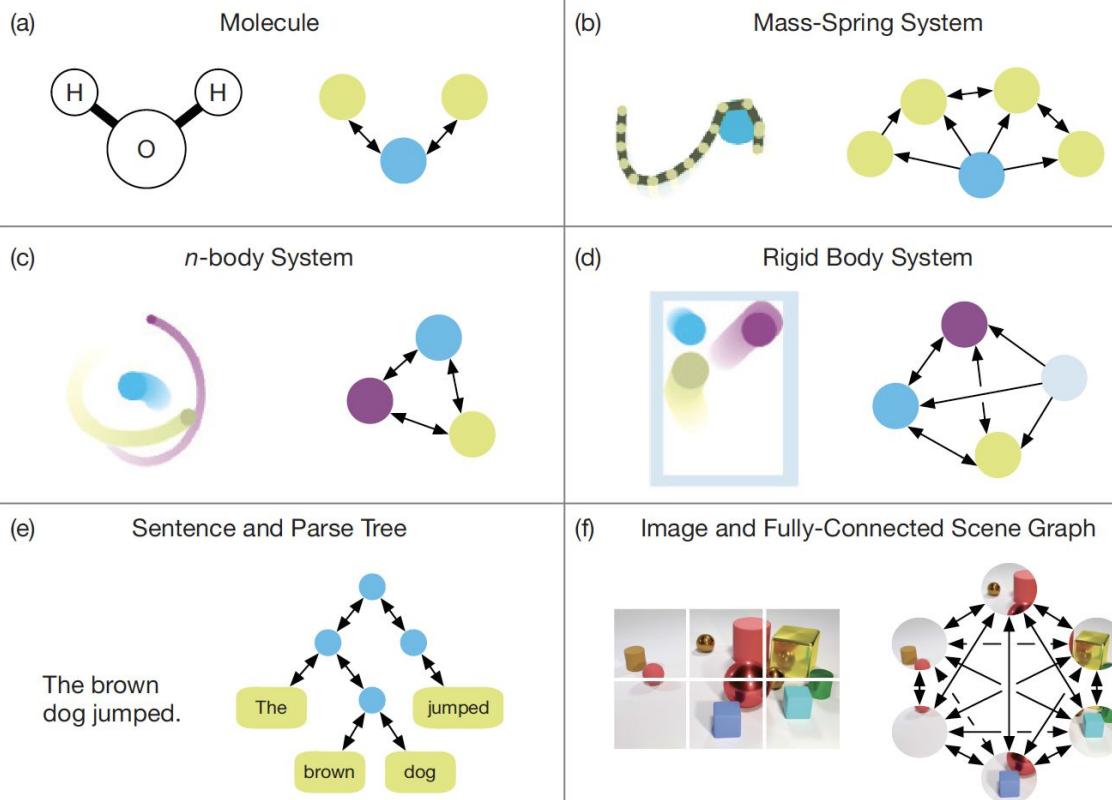


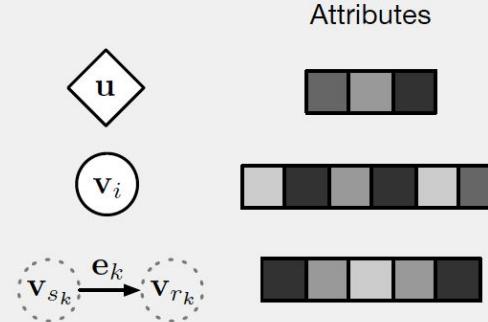
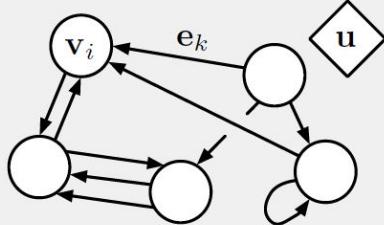
Figure 1: Reuse and sharing in common deep learning building blocks. (a) Fully connected layer, in which all weights are independent, and there is no sharing. (b) Convolutional layer, in which a local kernel function is reused multiple times across the input. Shared weights are indicated by arrows with the same color. (c) Recurrent layer, in which the same function is reused across different processing steps.

Different graph representations



Definition of ‘extended graph’

Box 3: Our definition of “graph”



Here we use “graph” to mean a **directed, attributed multi-graph with a global attribute**. In our terminology, a node is denoted as v_i , an edge as e_k , and the global attributes as u . We also use s_k and r_k to indicate the indices of the sender and receiver nodes (see below), respectively, for edge k . To be more precise, we define these terms as:

Directed : one-way edges, from a “**sender**” node to a “**receiver**” node.

Attribute : properties that can be encoded as a vector, set, or even another graph.

Attributed : edges and vertices have attributes associated with them.

Global attribute : a graph-level attribute.

Multi-graph : there can be more than one edge between vertices, including self-edges.

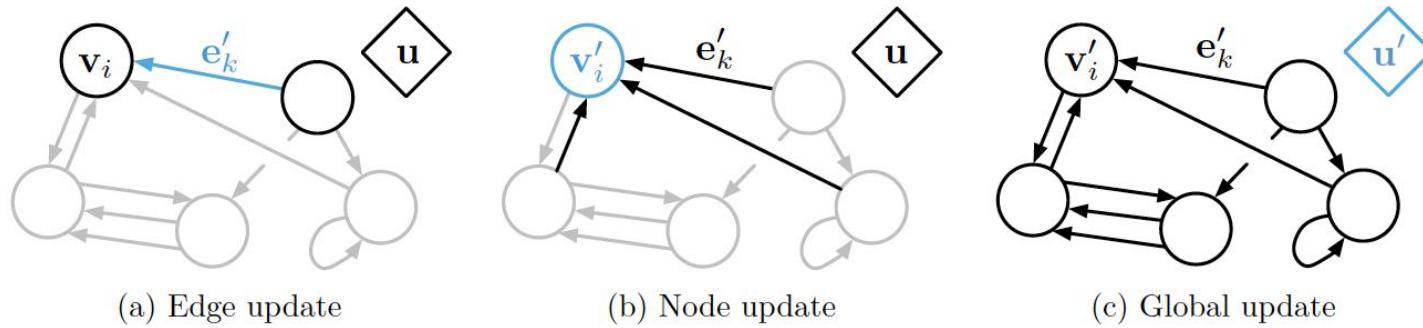
Figure 2 shows a variety of different types of graphs corresponding to real data that we may be interested in modeling, including physical systems, molecules, images, and text.

Signal propagation algorithm

Algorithm 1 Steps of computation in a full GN block.

```
function GRAPHNETWORK( $E, V, \mathbf{u}$ )
    for  $k \in \{1 \dots N^e\}$  do
         $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$                                 ▷ 1. Compute updated edge attributes
    end for
    for  $i \in \{1 \dots N^n\}$  do
        let  $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$ 
         $\bar{\mathbf{e}}'_i \leftarrow \rho^{e \rightarrow v}(E'_i)$                                          ▷ 2. Aggregate edge attributes per node
         $\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$                                ▷ 3. Compute updated node attributes
    end for
    let  $V' = \{\mathbf{v}'\}_{i=1:N^v}$ 
    let  $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$ 
     $\bar{\mathbf{e}}' \leftarrow \rho^{e \rightarrow u}(E')$                                          ▷ 4. Aggregate edge attributes globally
     $\bar{\mathbf{v}}' \leftarrow \rho^{v \rightarrow u}(V')$                                          ▷ 5. Aggregate node attributes globally
     $\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$                                ▷ 6. Compute updated global attribute
    return  $(E', V', \mathbf{u}')$ 
end function
```

Updates in a GN block



Blue indicates the element that is being updated, and black indicates other elements which are involved in the update.

- The pre-update value of the blue element (previous time step) is also used in the update.

Summary

- A rather conceptual work.
 - No experimental validation.
- Unifying perspective
- The authors show how a few major earlier architectures can be realized within the GN framework:
 - Message-passing NN
 - Non-local neural networks (NLNN)
 - Relation Networks
 - Deep Sets
 - PointNet (see next slides)
 - ...

Summary

- Deep learning offers a wide repertoire of methods for learning from graphs and processing graphs.
- Models can learn to canonicalize representations and make them invariant with respect to various transformations.
- Continuing work on the theoretical underpinnings.

Recommended reading:

- Introductory lecture on Geometric Deep Learning: M. Bronstein, "Geometric Deep Learning: The Erlangen Programme of ML",
<https://www.youtube.com/watch?v=w6Pw4MOzMu0>
- A library for graph storage and operations: <https://networkx.org/>
- The PyTorch Geometric library: see next slide.

Recommended library (as of 2021/22)

PyTorch Geometric (PyG)



It consists of various methods for deep learning on graphs and other irregular structures, also known as geometric deep learning, from a variety of published papers. In addition, it consists of an easy-to-use mini-batch loader for many small and single giant graphs, multi gpu-support, a large number of common benchmark datasets (based on simple interfaces to create your own), and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds.

Module 7: Models for processing point clouds

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Outline

1. [Introduction](#): the concept of point cloud.
2. Models for PC classification and segmentation: [PointNet](#)
3. Models for PC completion: [PointTr](#)

Introduction

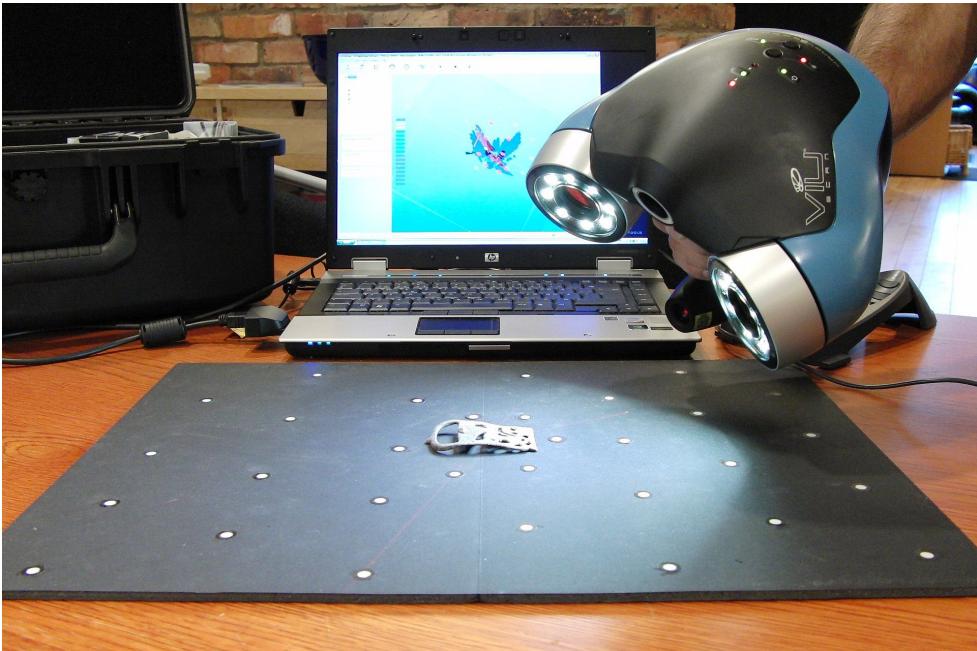
The concept of point cloud

- The previous module was about graph neural networks (GNN).
What if I just have a set, not a graph?
- Set is a graph without edges. But it may still form some structure, e.g. by virtue of the attributes of individual elements.
- It is thus worth considering two cases:
 - Unstructured, generic sets: there are dedicated models, e.g., Deep Sets [1], Set Transformers [2], and others – not covered in this module.
 - Structured sets, the special case of which are point clouds:
 - Point cloud = a set of points in geometric (usually 3D Cartesian) space.
 - In other words, the attributes of set elements embed them in a Cartesian space.

[1] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, Alexander Smola, Deep Sets, <https://arxiv.org/abs/1703.06114>

[2] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, Yee Whye Teh, Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks, <https://arxiv.org/abs/1810.00825>

Examples of 3D laser scanners



Credits: Wikipedia https://en.wikipedia.org/wiki/3D_scanning

Point clouds

Characteristics:

- No intrinsic ordering (set)
- Variable number of points
- Typically large number of points
- Individual points may have no features themselves (except coordinates)
- Rotational invariance is desirable
- Notable presence of noise (false positives, imprecise measurements, etc.)

Hence, learning from point clouds typically requires rotational (in 3D) and permutation invariance, and resistance to noise, meant as

- some points being FNs or FPs, or/and
- errors in point coordinates.

Types of common tasks related to point clouds

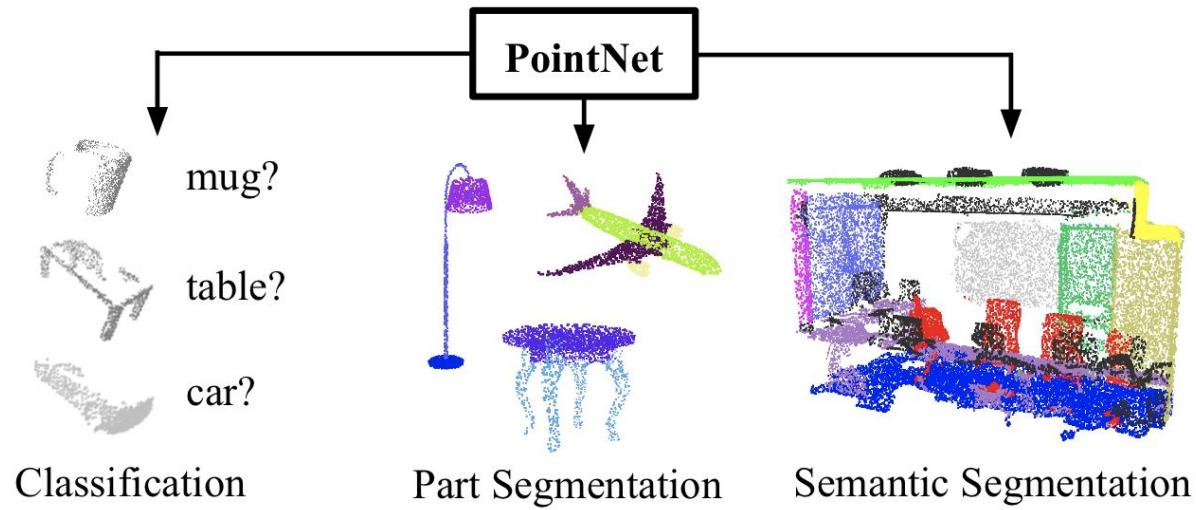
- Classification: object recognition based on a (often incomplete) PC
- Segmentation: partitioning of a PC into subsets that correspond to semantic categories (usually objects).
- Completion: completing imperfect PCs.

PointNet

PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation

Charles R. Qi, Hao Su, Kaichun Mo, Leonidas J. Guibas

<https://arxiv.org/abs/1612.00593>



Key ideas

Up to certain stage of processing, each point is processed independently.

Two main stages:

1. Use Spatial Transformer Network (Jaderberg et al., 2015) to predict an ‘optimal’ 3D rotation matrix and rotate the points (the key idea)

- The model can learn a ‘canonical’ representation of a point cloud
- The architecture of STN is similar to a simplified PointNet

2. Aggregate the ‘descriptors’ (features, ‘encoding’) of individual points with a transformation that is permutation-invariant (symmetric) and can be applied to an arbitrary number of points

- Technically: maxpooling

The result: a latent vector of 1024 dimensions, summarizing the entire input cloud

Aggregation scheme

Mathematical formulation of permutation-invariant processing of sets:

$$f(\{x_1, \dots, x_n\}) \approx g(h(x_1), \dots, h(x_n))$$

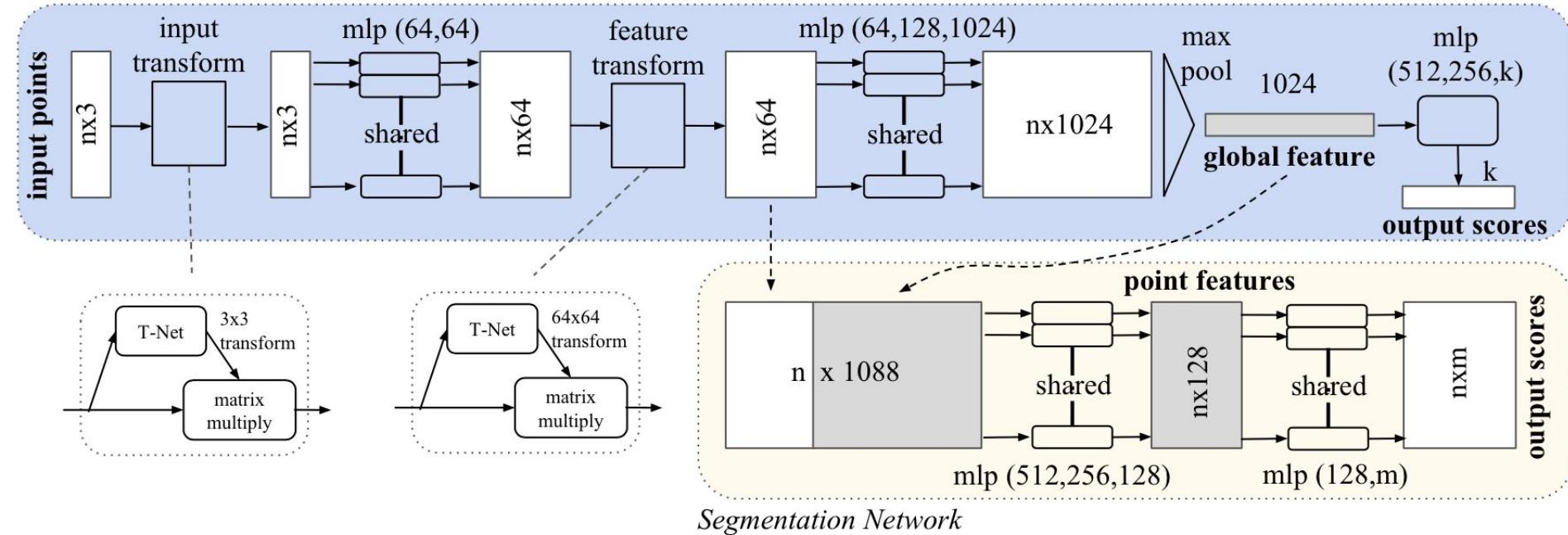
where

- f: function to be approximated
- x_i : points in the 3D cloud
- g: symmetric (permutation-invariant) function
- h: function processing each point individually

Many models (including PointNet) meet the above requirement exactly.

PointNet architecture

Classification Network



T-Net = STN; n: the number of points in the cloud

- Minor: STN is even easier to apply here than in (Jadeberger et al): no sampling grid needed!

Comments

- The first MLP ‘uplifts’ point representation from 3 to 64 dimensions ('descriptors').
- The second T-Net (STN) transform operates in that 64D space.
- Final processing stage:
 - For classification: a dense decoder with layer sizes of 512, 256, and k maps from the latent space (global feature vector) to k decision classes.
 - For segmentation: descriptors of individual points concatenated with the latent (global feature), and then each point keeps being processed independently, and finally classified into one of target regions.

Tasks

Tasks considered:

- classification: point cloud → class label
- part segmentation: partitioning of a point cloud
- semantic segmentation: partitioning of a point cloud

For the classification task:

- Benchmark ModelNet40: 12,311 CAD models (9,843 train, 2,468 test) from 40 man-made object categories.
- 1024 points sampled from each example (from the set of available measurements) and normalized to the unit sphere.
- Augmentations: random rotation around vertical axis; Gaussian jitter added to each point.

The classification task: Results

	input	#views	accuracy avg. class	accuracy overall
SPH [11]	mesh	-	68.2	-
3DShapeNets [28]	volume	1	77.3	84.7
VoxNet [17]	volume	12	83.0	85.9
Subvolume [18]	volume	20	86.0	89.2
LFD [28]	image	10	75.5	-
MVCNN [23]	image	80	90.1	-
Ours baseline	point	-	72.6	77.4
Ours PointNet	point	1	86.2	89.2

The first paper that approaches this problem with just points (not volumetric, not multi-view)

Semantic segmentation



Top row: input point cloud superimposed on the 2D image of the scene.
Bottom row: the result of semantic segmentation.

Semantic segmentation

	mean	aero	bag	cap	car	chair	ear phone	guitar	knife	lamp	laptop	motor	mug	pistol	rocket	skate board	table
# shapes		2690	76	55	898	3758	69	787	392	1547	451	202	184	283	66	152	5271
Wu [27]	-	63.2	-	-	-	73.5	-	-	-	74.4	-	-	-	-	-	-	74.8
Yi [29]	81.4	81.0	78.4	77.7	75.7	87.6	61.9	92.0	85.4	82.5	95.7	70.6	91.9	85.9	53.1	69.8	75.3
3DCNN	79.4	75.1	72.8	73.3	70.0	87.2	63.5	88.4	79.6	74.4	93.9	58.7	91.8	76.4	51.2	65.3	77.1
Ours	83.7	83.4	78.7	82.5	74.9	89.6	73.0	91.5	85.9	80.8	95.3	65.2	93.0	81.2	57.9	72.8	80.6

Metric: mIoU (mean Intersection over Union, %) on cloud points.

Other contributions

- Comparison with alternative order-invariant methods:
Examined various ways of achieving rotational invariance (MLP on unsorted data, MLP on sorted data, LSTM, Sum, Avg, Max pooling).
 - Max pooling clearly best.
- Assessment of robustness:
 - Deletion of points
 - Insertion of points
 - Perturbation
- Low time and space complexity

	accuracy
MLP (unsorted input)	24.2
MLP (sorted input)	45.0
LSTM	78.5
Attention sum	83.0
Average pooling	83.8
Max pooling	87.1

Proof of universal approximation

Theorem 1. Suppose $f : \mathcal{X} \rightarrow \mathbb{R}$ is a continuous set function w.r.t Hausdorff distance $d_H(\cdot, \cdot)$. $\forall \epsilon > 0$, \exists a continuous function h and a symmetric function $g(x_1, \dots, x_n) = \gamma \circ \text{MAX}$, such that for any $S \in \mathcal{X}$,

$$\left| f(S) - \gamma \left(\text{MAX}_{x_i \in S} \{h(x_i)\} \right) \right| < \epsilon$$

where x_1, \dots, x_n is the full list of elements in S ordered arbitrarily, γ is a continuous function, and MAX is a vector max operator that takes n vectors as input and returns a new vector of the element-wise maximum.

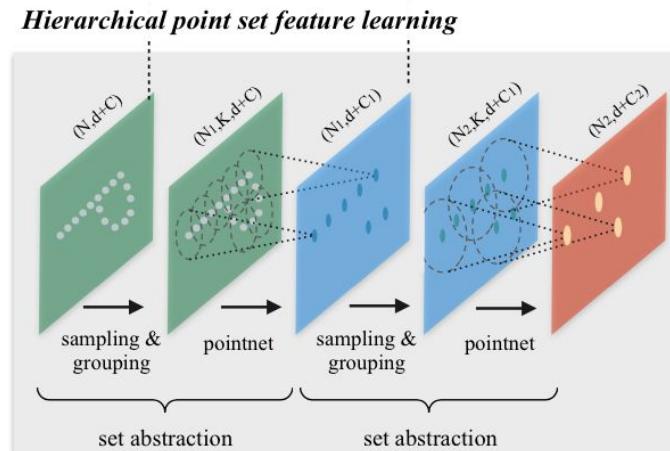
Follow-ups

Many other studies, including:

PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space
Charles R. Qi, Li Yi, Hao Su, Leonidas J. Guibas
<https://arxiv.org/abs/1706.02413>

Increments:

- Hierarchical grouping of points
- Sampling of most informative points in clouds
- Working with distances between points,
rather than coordinates.
 - No need to explicitly ensure rotational invariance!

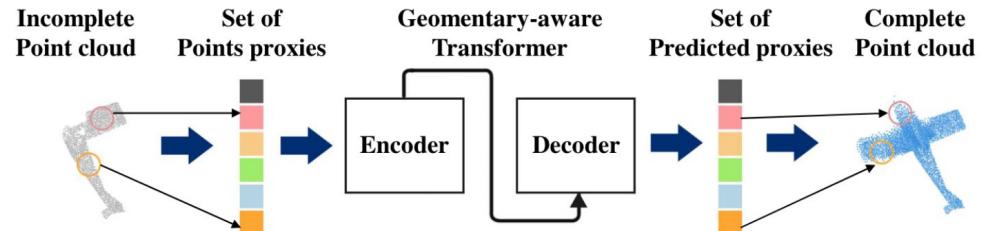
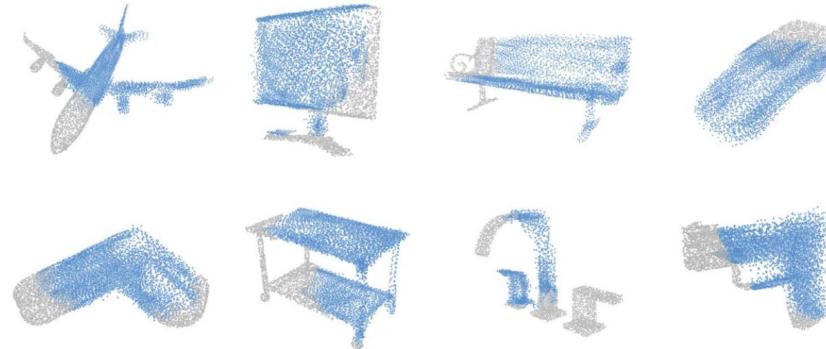


PointTr

PoinTr: Diverse Point Cloud Completion with Geometry-Aware Transformers

Xumin Yu, Yongming Rao, Ziyi Wang, Zuyan Liu, Jiwen Lu, Jie Zhou

<http://arxiv.org/abs/2108.08839>



The task: Point Cloud Completion

Point cloud data from 3D sensors are incomplete because of:

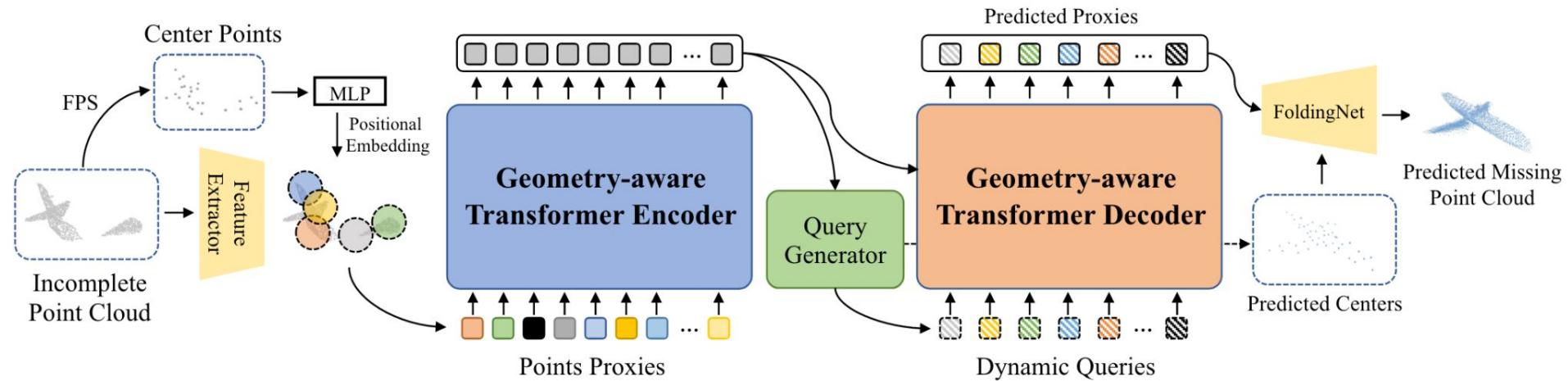
- occlusion,
- self-occlusion (one part of an object occluded by another part),
- light reflection,
- limited sensor resolution.

The goal: recover a complete point cloud from partial and/or sparse samples of points.

Note: Prior work to PointTr exists and has been reviewed in this paper.

Key aspects of the architecture

1. Based on the Transformer architecture.
 - a. More specifically: *two set-to-set transformers: encoder transformer and decoder transformer*
2. Processes *point proxies* (subclouds of points) rather than individual points.
 - a. Recall that a Transformer in general involves quadratic number of query-key interactions.
3. Queries on point proxies are realized in geometry-aware way.



Point proxy

Introduced to avoid quadratic number of query-key interactions (inacceptable given the sizes of typical point clouds).

1. Identify a fixed number N of point centers $\{q_1, \dots, q_N\}$ using the furthest point sample (FPS) method.
2. Apply a dynamic graph CNN (DGCNN [1]) with hierarchical downsampling to extract features F'_i of point centers.
3. Combine features of point centers F'_i with the local characteristics $\varphi(q_i)$ of q_i , obtained using a small MLP ($\varphi()$ can be likened to *positional embedding*).

$$F_i = F'_i + \varphi(q_i),$$

[1] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. TOG, 38(5):1–12, 2019.

Furthest point sampling

Two working sets: sampled S (initially empty) and remaining R (initially all the points).

1. For each point in x in S find its nearest neighbour y in R . Let its distance be $d(y)$.
2. Select the point y in R with largest $d(y)$ and move it from to S .
3. Repeat until a desired number of points in S has been collected.

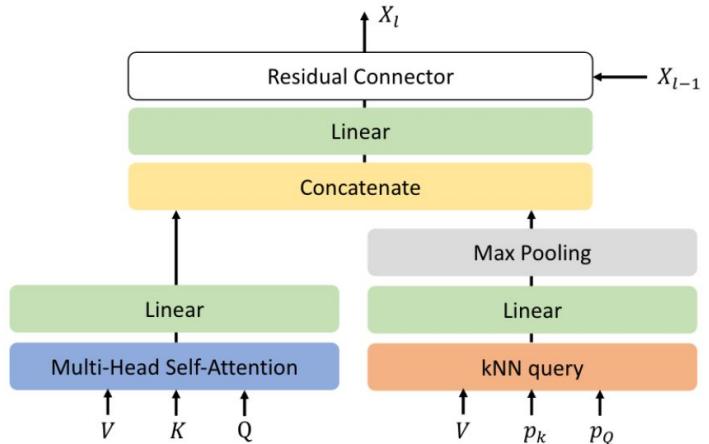
Geometry-aware querying

In contrast to conventional transformer, conducts two queries in parallel:

1. Conventional query Q to keys K .
2. Geometric query based on the k -nearest neighbors based on the coordinates p_Q of the query point proxy and coordinates p_k of the key point proxy.

Block 1 is based on feature similarity, while
block 2 on geometric similarity.

Details in the paper.

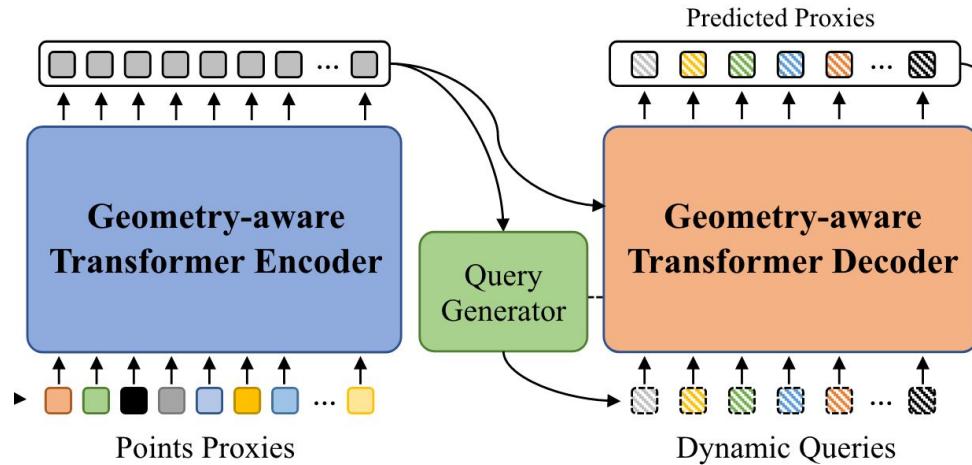


Dynamic queries

Given set of point proxies $F=\{F_1, \dots, F_N\}$, the overall processing can be described as:

$$\mathcal{V} = \mathcal{M}_E(\mathcal{F}), \quad \mathcal{H} = \mathcal{M}_D(Q, \mathcal{V}),$$

resulting in H , which is a set of M point proxies for the missing point cloud (note that M is general unrelated to N), where \mathcal{M}_E is the encoder transformer, \mathcal{M}_D is the decoder transformer, V is a set of N output features, and Q is a set of M dynamic queries .



Query generator

Query
Generator

Procedure inspired by [1]:

1. Summarize V with a linear project to higher number of dimensions.
2. Apply max pooling in the resulting space.
3. Apply linear projection layer to generate $3M$ features.
4. Reshape the resulting vector to 3D coordinates of M points $\{c_1, \dots, c_M\}$.
5. Concatenate the coordinates of each point with the global feature vector (probably the vector resulting from 2; not sure here),
6. Pass each resulting vector independently through a MLP, resulting in Q_i s
7. The resulting set is Q .

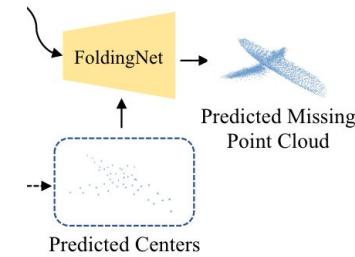
My comment: steps 3-4 is questionable: a linear *layer* produces a *set* of queries.

Projecting resulting point proxies to points

(what the authors call “Multi-Scale Point Cloud Generation”)

1. Uses the coordinates of queries c_i as reference points, i.e. local centers of missing sub-clouds.
2. Apply FoldingNet [1] independently to each output token H_i produced by the decoder transformer to obtain local displacements with respect to the center.
3. Combine these two, obtaining the resulting local sub-cloud P_i :

$$\mathcal{P}_i = f(H_i) + c_i, \quad i = 1, 2, \dots, M.$$



[1] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. Foldingnet: Interpretable unsupervised learning on 3d point clouds. CoRR, abs/1712.07262, 2017

Loss function

Chamfer distance (permutation-invariant) applied independently to:

- The local centers c (earlier: c_i) posited by the queries and the points g in the ground-truth (target) point cloud G (J_0), and
- The points p posited by the queries and the points g in the the ground-truth (target) point cloud G (J_1)

$$J_0 = \frac{1}{n_C} \sum_{c \in C} \min_{g \in G} \|c - g\| + \frac{1}{n_G} \sum_{g \in G} \min_{c \in C} \|g - c\|,$$

$$J_1 = \frac{1}{n_P} \sum_{p \in P} \min_{g \in G} \|p - g\| + \frac{1}{n_G} \sum_{g \in G} \min_{p \in P} \|g - p\|.$$

The final loss is $J_0 + J_1$.

Results

Visualization of point proxies:

Grey: input

Red: point proxies

Blue: completed point clouds

Yellow: ground truth



Results

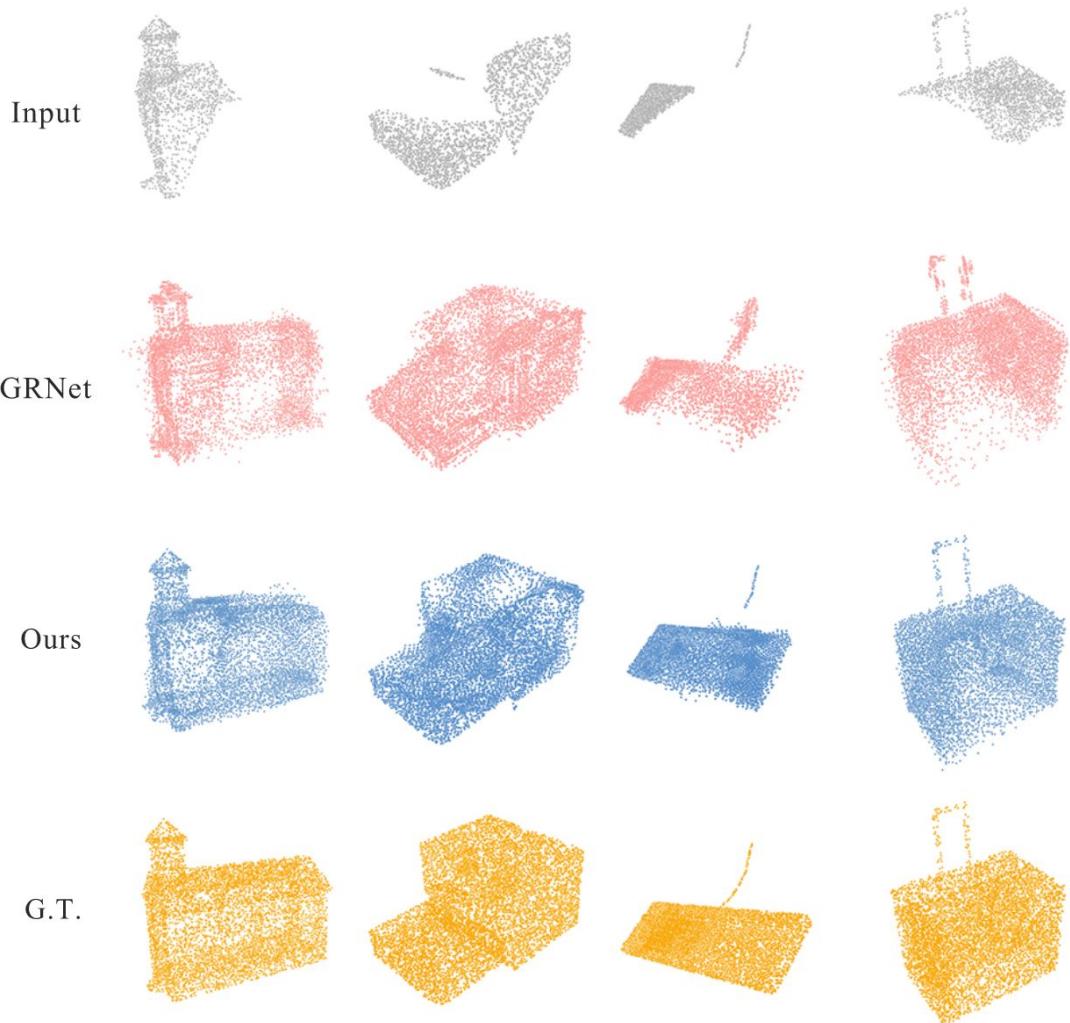
Systematically better than 5+ SotA methods.

Metric: a variant of Chamfer distance.

$$d_{CD}(\mathcal{P}, \mathcal{G}) = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \min_{g \in \mathcal{G}} \|p - g\| + \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} \min_{p \in \mathcal{P}} \|g - p\|$$

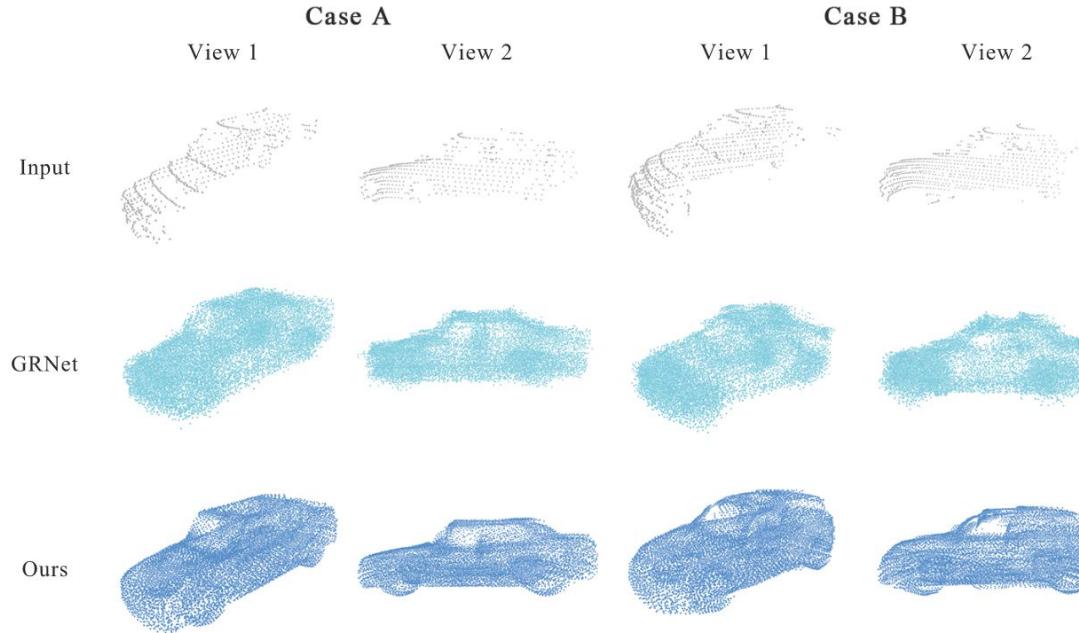
The figure: completion for objects from novel categories.

(Shape-34 dataset)



Multi-view experiments

The model is provided with 2 views of the same object .
(KITTI dataset, real-world LiDAR scans)



Module 8: Transformers and self-supervised learning

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Transformer models

Attention Is All You Need

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin

<https://arxiv.org/abs/1706.03762>

Citations (9999)

Data provided by:
[\(report data issues\)](#)



Filter: Sort: Influence ▼
Pages: ◀ 1 2 3 4 5 ... 1000 ▶ Skip: 1 ▼

The concept of transformer

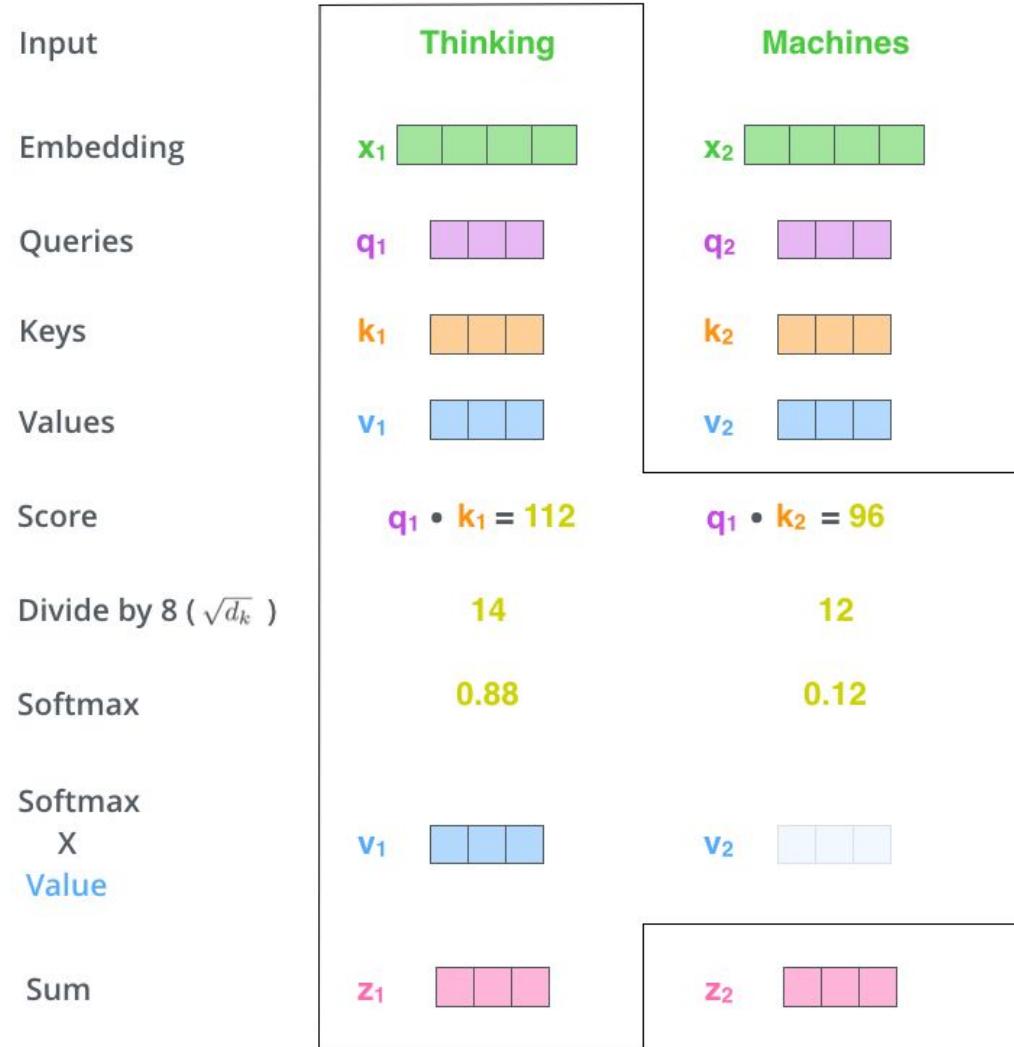
A (quite) general methodology for processing composite inputs (sequences, trees, graphs, ...).

Key ideas:

1. Define a (quite sophisticated) context (neighborhood) when processing a given (i^{th}) element of the input (often called token).
2. Make the selection of context trainable (trainable attention).
 - o So-called self-attention.
 - o Achieved using the mechanism of queries (q) and keys (k).
3. Detach the signal (value) from the attention.

Visual explanation

The part in black frame describes the processing for the first token (x_1). It involves sending queries to all tokens in the neighborhood, including the same token.



Credits:

<https://jalammar.github.io/illustrated-transformer/>

Steps of processing

From the perspective of i^{th} element of the input:

1. Input embedding; produces x_i for i^{th} element of the input.
 2. Calculation of query q_i , key k_i , and value v_i (d -dimensional vectors) by multiplying x_i by three matrices: W^Q , W^K , and W^V .
 3. Calculate the scalar score $s_{ij} = q_i \cdot k_j$ (dot product) for each combination of query and key.
 - a. Recall that dot product (for normalized vectors) is equivalent to cosine distance.
 4. Normalize the scores $s_{ij} \leftarrow s_{ij} / \sqrt{d}$ and pass them through softmax $s_{*j} \leftarrow \text{softmax}(s_{*j})$ (softmax calculated over j)
 5. Weigh the values by scores: $z_{ij} \leftarrow s_{ij} v_j$ and sum them $z_i \leftarrow \sum_j z_{ij}$
- The resulting z_i s are then passed independently through an ‘postprocessing’ layer.

Steps 2-5 map x_i to z_i . The above processing describes a single attention head.

Self-attention: the cornerstone of transformers

The interactions between queries and keys are essential in transformers.

This mechanism is called self-attention.

One-liner formula using matrix notation:

$$S = \text{softmax} \left(\frac{QK^T}{\sqrt{d_q}} \right)$$

where:

- S : $n \times n$ attention matrix
- Q : $n \times d_q$ matrix of queries
- K : $n \times d_q$ matrix of keys
- d_q : dimensionality of queries and keys
- n : size of the neighbourhood/context

Remarks

For sequences, i represents the index of the element.

- However, in general it is some position of the element in the composite input.
This could be alternatively:

- The position of a patch of a raster image,
- The index of a child node in a tree,
- The index of a neighboring node in a graph,
- ...

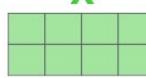
Most transformers use multiple attention heads.

- The resulting z_i 's are concatenated (and optionally projected linearly using another matrix) and fed into further processing.
- The next slide presents this,
 - however, abstracting from i (the index of sequence elements), hence the z s produced by particular heads are matrices, and not vectors (z is stacked into matrices).

Multiple attention heads: visual explanation

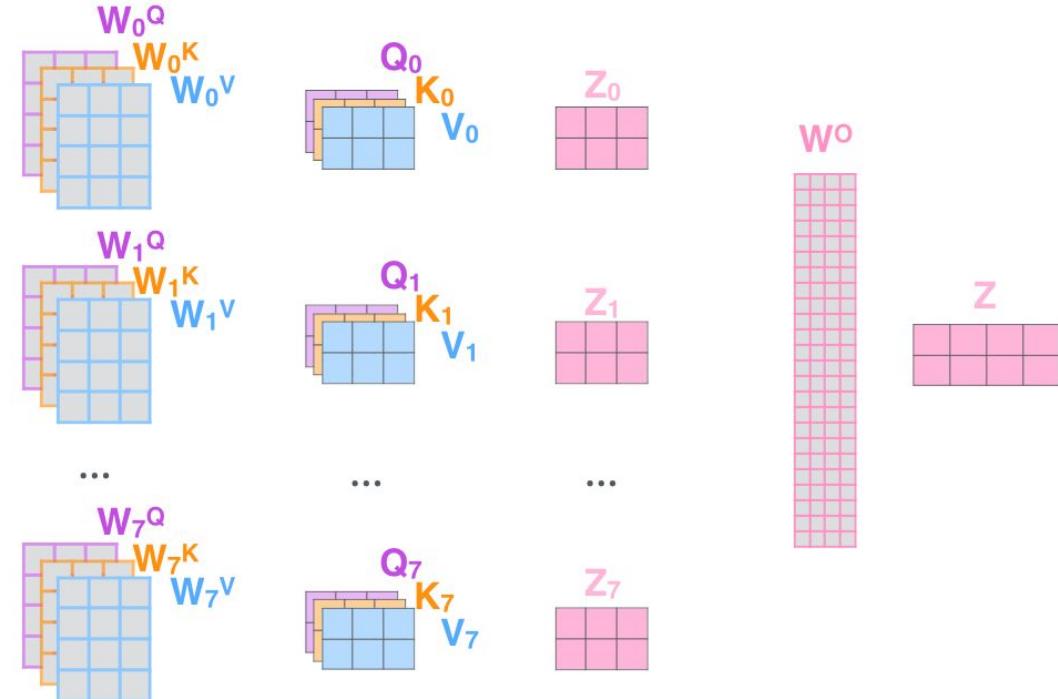
- 1) This is our input sentence* X
- 2) We embed each word*
- 3) Split into 8 heads. We multiply X or R with weight matrices
- 4) Calculate attention using the resulting $Q/K/V$ matrices
- 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer

Thinking
Machines



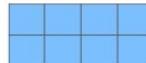
X

A green 4x4 grid representing the input sentence "Thinking Machines".



* In all encoders other than #0, we don't need embedding.
We start directly with the output of the encoder right below this one

R



A blue 4x4 grid representing the input matrix R .

Positional encoding

- **Motivation:** The neighborhood/window is uniform, has no structure.
- **The goal:** To inform the model about the location of tokens in the sequence.
- **The means:** An artificial ‘signal’ that provides that information.
- **Question 1:** How to encode the information about the position?
 - Naive indexing of the sequence with integer would not work well – DL models are not good at interpreting such information.
 - Therefore: positional encoding has the form of multi-dimensional vectors, with elements changing periodically along the sequence.
 - Typical formulation: sine and cosine functions of various periodicity:

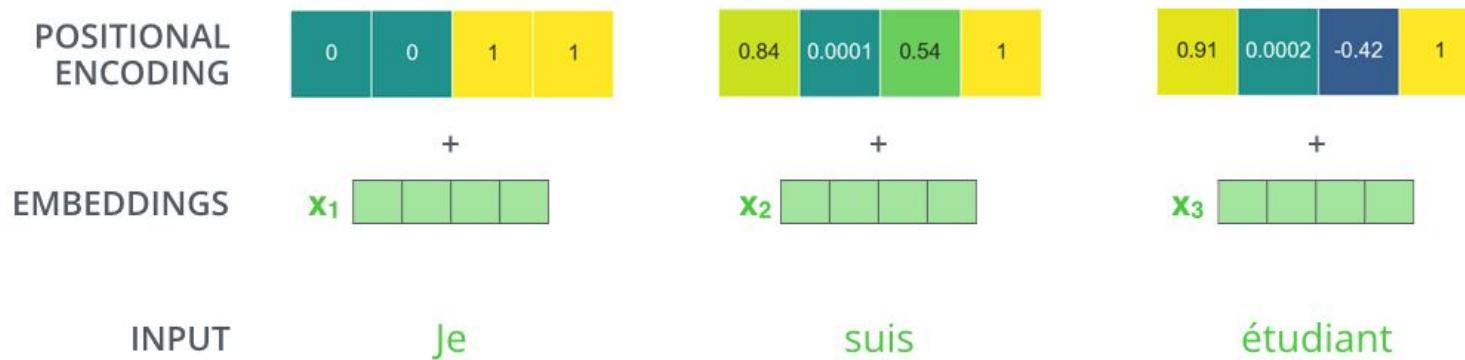
$$\text{PE}_{pos, 2i} = \sin(pos / 10,000^{2i / d_{model}}) \quad \text{PE}_{pos, 2i + 1} = \cos(pos / 10,000^{2i / d_{model}})$$

where: d_{model} is vector dimensionality, i is the index in the vector, pos is the position.

- Works for arbitrary long sequences.
- Typically based on absolute positions (less often as relative positions within the window).

Positional encoding

- **Question 2:** How to convey that information to the model?
 - The common approach: the positional encoding vectors are simply added to the input (i.e. to tokens' embeddings).
- Example:



- Advantage: no additional inputs required.
- However, alternative methods have been proposed too.

Transformers: Pros and cons

Advantages, in particular in comparison to RNNs, which used to be the primary architectures for processing sequences:

- Less performance degradation when sequences get longer.
- Truly parallel processing
 - Faster training and querying (depends on the use case, but almost always true).
- More general model
 - More flexible definition of the context, can help at ‘forgetting’ of older elements in the sequence.
 - Separation of value (*what* should be passed) from key (*which element* should be passed).
 - Can be conveniently generalized to other data types that feature a natural notion of window/context/neighborhood; for instance:
 - raster images,
 - graphs,
 - point clouds,
 - ...

Transformers: Pros and cons

Disadvantages:

- Tempting to use large numbers of parameters, and thus overfitting.
- Quadratic complexity of the query-key interactions:
 - The extent of the window n (often referred to ‘the number of input tokens’) translates into $O(n^2)$ interactions between queries and keys.
 - For contemporary transformers, n is quite often in the order of hundreds, or even thousands.
 - This is the main determinant of the high computational cost of large-scale transformers (in particular LLMs).
 - This challenge affects both training and querying.
 - See next slides for an example of a possible remedy.

Addressing the quadratic complexity of transformers

A notable example:

Nyströmformer: A Nyström-Based Algorithm for Approximating Self-Attention

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, Vikas Singh

<https://arxiv.org/abs/2102.03902>

- Key idea: use a principled known method to approximate the calculation of the products between queries and keys.
- From the abstract:

Our idea is based on adapting the Nyström method to approximate standard self-attention with $O(n)$ complexity. The scalability of Nyströmformer enables application to longer sequences with thousands of tokens.

The gist of the idea

Denote the softmax matrix used in self-attention $S = \text{softmax} \left(\frac{QK^T}{\sqrt{d_q}} \right) \in \mathbf{R}^{n \times n}$. S can be written as

$$S = \text{softmax} \left(\frac{QK^T}{\sqrt{d_q}} \right) = \begin{bmatrix} A_S & B_S \\ F_S & C_S \end{bmatrix}, \quad (3)$$

where $A_S \in \mathbf{R}^{m \times m}$, $B_S \in \mathbf{R}^{m \times (n-m)}$, $F_S \in \mathbf{R}^{(n-m) \times m}$ and $C_S \in \mathbf{R}^{(n-m) \times (n-m)}$. A_S is designated to be our sample matrix by sampling m columns and rows from S .

Quadrature technique. S can be approximated via the basic quadrature technique of the Nyström method. It begins with the singular value decomposition (SVD) of the sample matrix, $A_S = U\Lambda V^T$, where $U, V \in \mathbf{R}^{m \times m}$ are orthogonal matrices, $\Lambda \in \mathbf{R}^{m \times m}$ is a diagonal matrix. Based on the out-of-sample columns approximation (Wang and Zhang 2013), the explicit Nyström form of S can be reconstructed with m columns and m rows from S ,

Training of transformers

- Most common training blueprint: self-supervision: given an example x (e.g. sequence):
 - Present the model with x ‘partially occluded’, i.e. with some tokens masked out.
 - Train the model to respond with complete x , i.e. with some form of reconstruction loss.
- Self-supervision can be viewed as autoassociation taken one step further:
 - Learn how to reconstruct the input, even if it is only partially visible.
- Forces the model to learn structural co-dependencies between the elements of the presented structure.
 - For instance, collocations of words in sentences or syllables in words.
- Technical realizations of masking:
 - Replacement with a fixed, ‘neutral’ embedding, like zero-vector.
 - Replacement with a trainable mask.

Notable pretrained transformer models

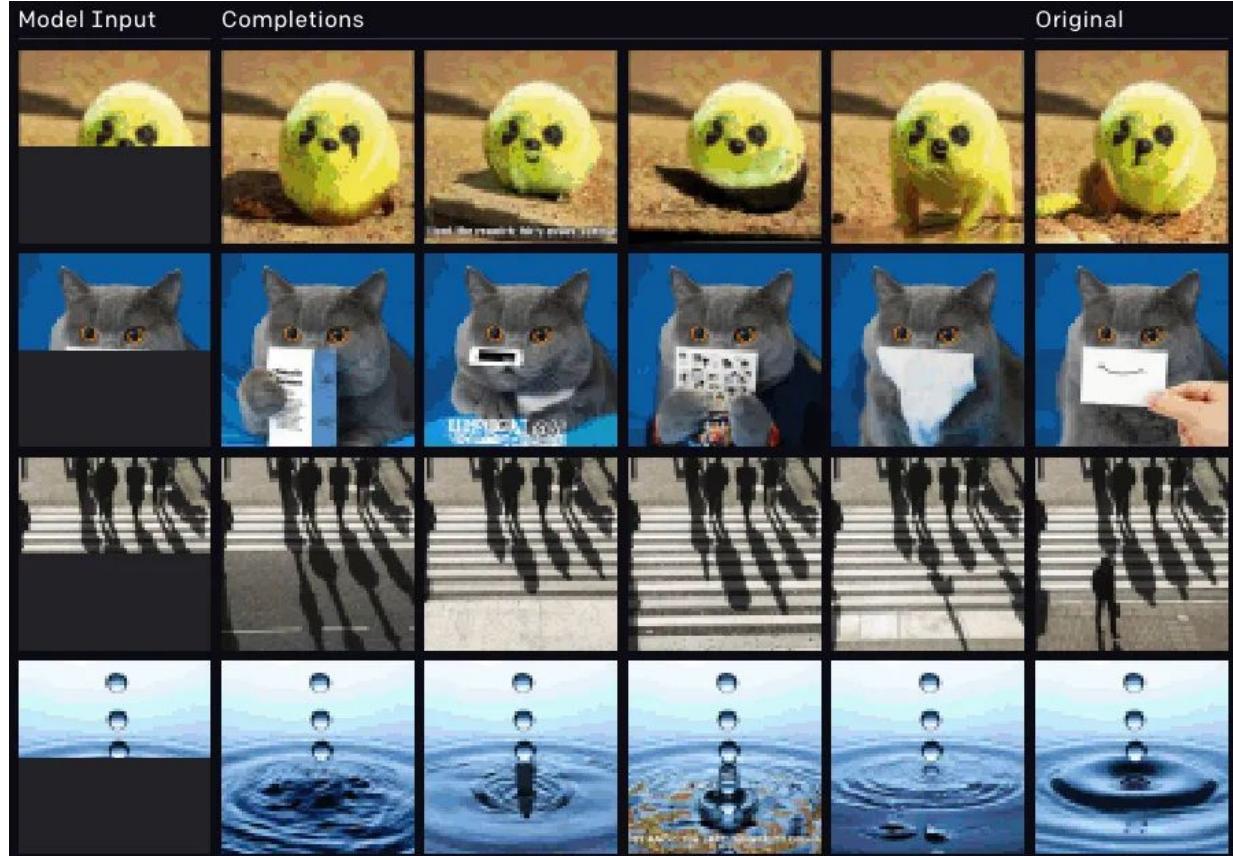
Most efforts focus on NLP:

- Bidirectional Encoder Representations from Transformers (BERT)
- Generative Pre-trained Transformer (GPT)
 - GPT-2
 - GPT-3
 - GPT-4
- Gemini
- Claude
- Microsoft Copilot
- Deepseek (January 2025)

Image completion

A natural task for transformers, given their training with self-supervision.

Example:
GPT-2 proposing various ways of completing the same input image.



ChatGPT



- Trained by OpenAI
- Language generation model
- Can answer questions, generate text, and understand natural language input
- Knowledge cutoff: 2021
- Continuously updating and improving abilities.

Fun facts:

- Single query cost: 0.02¢, several (4-5?) orders of magnitude larger than
- OpenAI spends about \$3M per day on the cloud that runs ChatGPT
- Technically owned by Microsoft (and uses MS Azure cloud as a backend)

The main differences between ChatGPT and GPT-2

- ChatGPT is fine-tuned for conversational tasks, while GPT-2 is a more general-purpose language model.
- GPT-2 has a larger model size than ChatGPT, with 1.5 billion parameters compared to 175 million parameters.
- GPT-2 has been trained on a diverse range of internet text, while ChatGPT has been fine-tuned on specific conversational tasks.
- ChatGPT has a specific use case for specific tasks in conversational AI.
- GPT-2 is more powerful and generalizable than ChatGPT because of its larger model size and diverse training data.

Example

This is how I obtained
the content for the
previous slide :D

The screenshot shows a web browser window with the URL chat.openai.com/chat. The page title is "ChatGPT Personal Slider". The interface includes a sidebar with "New chat", "ChatGPT Personal Slider", "Clear conversations", "Dark mode", "OpenAI Discord", and "Updates & FAQ". The main area is a chat log:

- K Hi ChatGPT, could you please create a one-slider about yourself?
- [AI] I'm sorry, I am a text-based AI model and do not have the capability to create visual presentations such as a one-slider. Is there anything else I can assist you with? thumb up thumb down
- K So could you please generate a one-slide textual summary of yourself?
- [AI] Sure, here's a one-slide summary of myself:
ChatGPT:
 - Trained by OpenAI
 - Language generation model
 - Can answer questions, generate text, and understand natural language input
 - Knowledge cutoff: 2021
 - Continuously updating and improving abilities.
- K Thank you. Could you list in bullets the main differences between you and GPT-2?
- [AI] Sure, here are the main differences between ChatGPT and GPT-2:
 - ChatGPT is fine-tuned for conversational tasks, while GPT-2 is a more general-purpose language model.
 - GPT-2 has a larger model size than ChatGPT, with 1.5 billion parameters compared to 175 million parameters.
 - GPT-2 has been trained on a diverse range of internet text, while ChatGPT has been fine-tuned on specific conversational tasks.
 - ChatGPT has a specific use case for specific tasks in conversational AI.
 - GPT-2 is more powerful and generalizable than ChatGPT because of its larger model size and diverse training data.

At the bottom right of the chat area are "Regenerate response" and "Send" buttons. The number "20" is located at the bottom right corner of the slide.

Are transformers/GPTs the ‘killer app’ of ML?

Arguably GTPs are quite unique in their capability to capture knowledge from a multitude domains.

- The same model can be asked questions about geography, math, etc.

However, that knowledge seems to be quite superficial (see the apt comment on the right-hand side).

Julian Togelius @togelius · Jul 17, 2020 
I have the same impression. We can now automate the production of passable text on basically any topic. What's hard is to produce text that doesn't fall apart when you look closely. But that's hard for humans as well.

Simon Sarris @simonsarris
GPT-3 imitating human text: We aren't pulling the mask off the machine to reveal a genius wizard, we're pulling the mask off each other to reveal the bar is low.

Julian Togelius @togelius
GPT-3 often performs like a clever student who hasn't done their reading trying to bullshit their way through an exam. Some well-known facts, some half-truths, and some straight lies, strung together in what first looks like a smooth narrative.

4:22 PM · Jul 17, 2020 
 171  Reply  Copy link
[Read 13 replies](#)



François Chollet ✅
@fchollet

...

LLMs = 100% memorization. There is no other mechanism at work.

A LLM is a curve fitted to a dataset (that is to say, a memory), with a sampling mechanism on top (which uses a RNG, so it can generate never-seen-before token sequences).

It doesn't just memorize and output back exact token sequences, obviously. It can memorize and reuse any pattern found in its training data, include certain categories of programs. fchollet.substack.com/p/how-i-think-...

But crucially, those patterns must have been featured in the training data. They need to have been memorized. LLMs break down on anything that wasn't in their training data. Like ARC. Because they're 100% memorization. Which makes sense, because a LLM is literally a curve fitted to some data points. What else could you expect?

Hilarious examples of chatbots' failures

= ChatGPT 4 > 

D You

Create a picture of an empty room with no elephant in it. Absolutely no elephant anywhere in the room.

ChatGPT



I've created another image of an empty room with no elephant in it. If there's anything more you need, feel free to ask!

New Folder
@ujmappa  ...

It's a bit addictive prompt, I especially like the text results with MS Designer / Copilot



draw me a kitchen with absolutely no polar bear in it

85

Explore ideas

Creations



Is more data the key to progress in AI (and AGI)?

(...) *The bitter lesson is based on the historical observations that*

- 1) *AI researchers have often tried to build knowledge into their agents,*
- 2) *this always helps in the short term, and is personally satisfying to the researcher, but*
- 3) *in the long run it plateaus and even inhibits further progress, and*
- 4) *breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning.*

(...) *One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are search and learning.*

(...) *We want AI agents that can discover like we can, not which contain what we have discovered.*

Module 9: Model optimization and related topics

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2024

<http://www.cs.put.poznan.pl/kkrawiec/>



Table of contents

1. [Slide 3: Architectural components](#)
2. [Slide 26: Model optimization](#)
3. [Slide 67: Other components and techniques](#)

Architectural components

Advanced activation functions

The baseline: ReLU

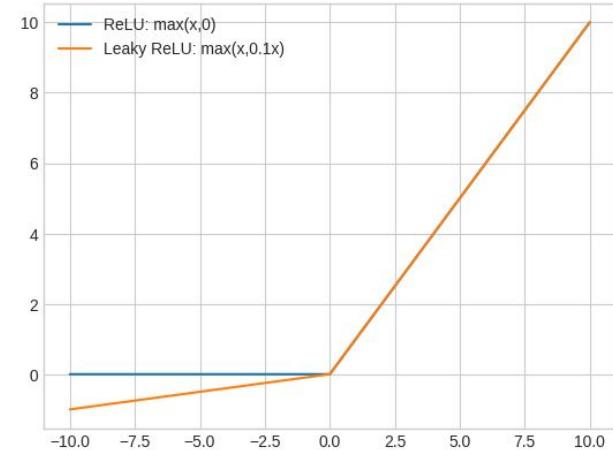
Fukushima, 1969 (the inventor of the Neocognitron, 1980).

The advantages:

- Better propagation of gradient
- Nonparametric
- Scale-invariant
- Computationally efficient
- Naturally gives rise to sparse representations

The disadvantages:

- ‘Dying ReLU’ phenomenon: once fed with a strongly negative excitation (argument), may get stuck there.
 - This problem is partially addressed by Leaky ReLU: $\max(x, 0.1x)$
- Coarse response surface.



Coarse response surface

Models featuring many (or exclusively) ReLU-like activation functions tend to have coarse response surfaces.

Many real-world use cases require smooth response surfaces; examples:

- regression,
- control.



Parametric Rectified Linear Unit – PReLU

Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun

<https://arxiv.org/abs/1502.01852>

- Learnable/parameterized generalization of Leaky ReLU

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}.$$

- Defined per output channel (index i).
- Can also realize:
 - ‘Regular’ ReLU: $a_i = 0$
 - Leaky ReLU: $a_i = 0.1$ or similar, fixed.

PReLU

- Addresses the ‘dying ReLU’ phenomenon.
- The steepness parameter is learned separately per channel.

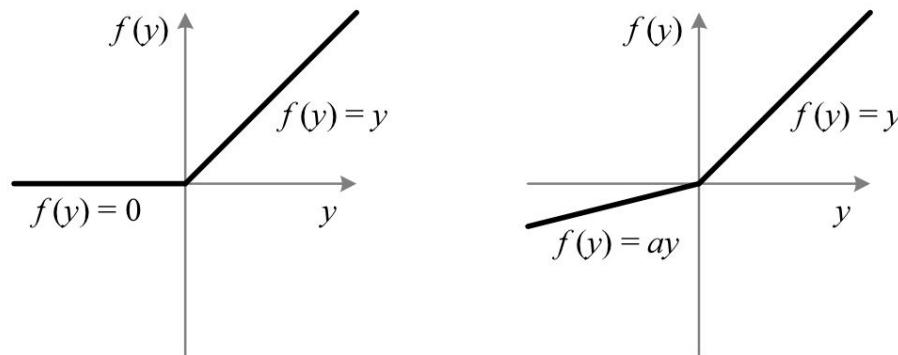


Figure 1. ReLU vs. PReLU. For PReLU, the coefficient of the negative part is not constant and is adaptively learned.

Swish activation function

Searching for Activation Functions

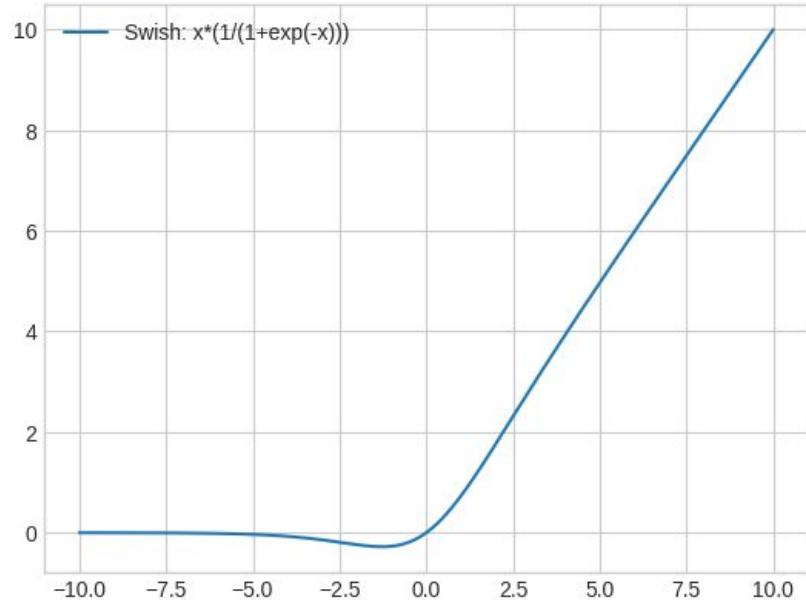
Prajit Ramachandran, Barret Zoph, Quoc V. Le

<https://arxiv.org/abs/1710.05941>

Swish activation function

Formula: $f(x) = x \text{sigmoid}(\beta x)$

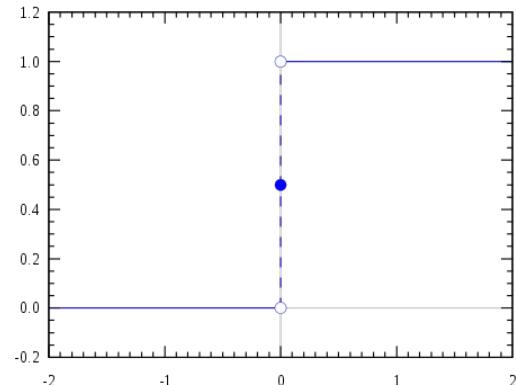
- **Self-gating:** the function *gates* its input with the sigmoid gate.
- For $\beta=1$: sigmoid linear unit, **Silu**, figure on the right [1].
 - However, β in Silu is constant.
- Replacing ReLUs improves top-1 classification accuracy on ImageNet by 0.9% for Mobile NASNet-A and 0.6% for Inception-ResNet-v2.



Swish activation function

- β ($\beta > 0$) is learnable – becomes a parameter of the model
 - For $\beta = 0$, Swish reduces to $x/2$
 - For large β , sigmoid approaches the step function (Heaviside function, below), which causes Swish to behave almost like ReLU
- Therefore, Swish interpolates *nonlinearly* between
 - a linear function
 - and the ReLU function.
- **Self-regularization:** the concavity of the function in the negative part of the domain discourages large weights.

$$H(x) := [x \geq 0]$$



Mish activation function

Mish: A Self Regularized Non-Monotonic Activation Function

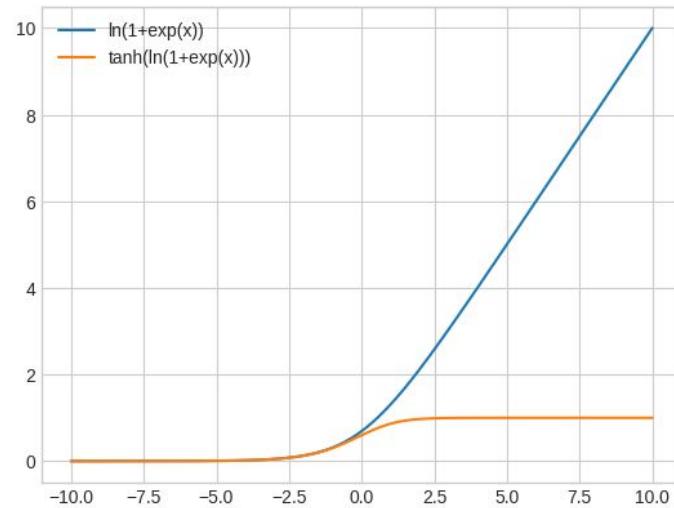
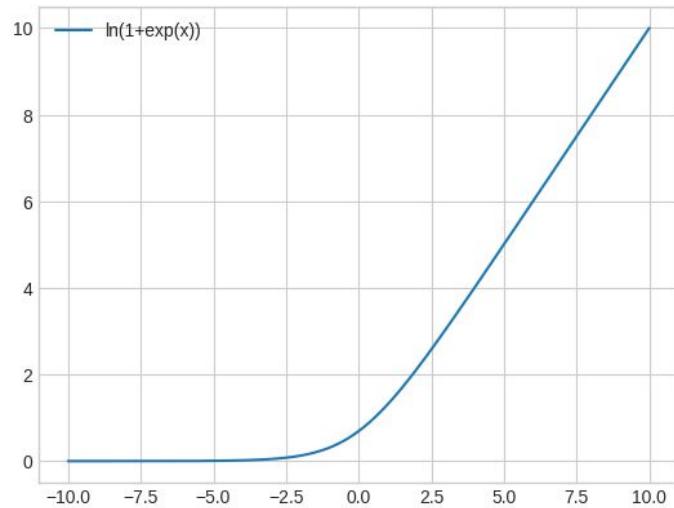
Diganta Misra

<https://arxiv.org/abs/1908.08681>

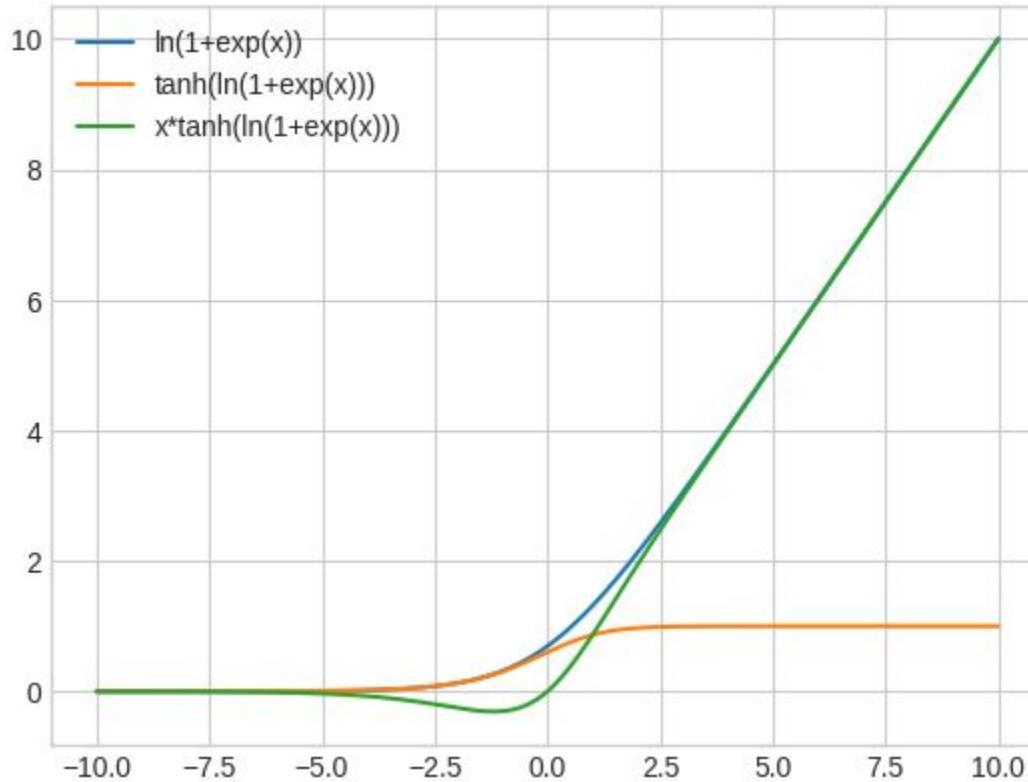
The concept of “self-gating”

Similarly to Swish, Mish uses the “*self-gating property*” where the non-modulated input is multiplied with the output of a non-linear function of the input.

$$f(x) = x \tanh(\text{softplus}(x)) = x \tanh(\ln(1 + e^x))$$

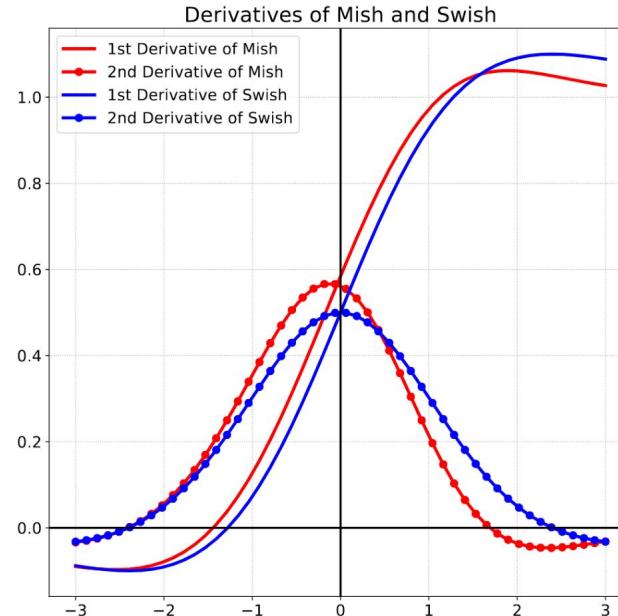
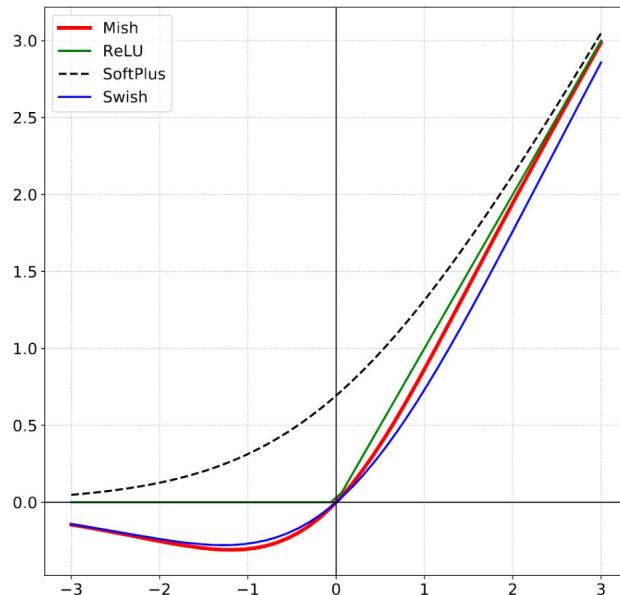


Mish



Mish activation function

“Distinctive negative concavity (unlike ReLU), which accounts for preservation of small negative weights”



Automating the synthesis of function formula

- The methodology used for both Swish and Mish: discovered by a Neural Architecture Search (NAS) over the space of the non-linear functions by a controlled search agent.
- Found by systematic analysis and experimentation over the characteristics that made Swish effective.
- When studying similarly behaved functions like Swish, which include $\arctan(x) \circ f_t \text{plus}(x)$, $\tanh(x) \circ f_t \text{plus}(x)$, $x \log(1 + \arctan(ex))$ and $x \log(1 + \tanh(ex))$, where $\circ f_t \text{plus}(x) = \ln(1 + ex)$, authors determined that Mish consistently outperforms the aforementioned functions along with Swish and ReLU.

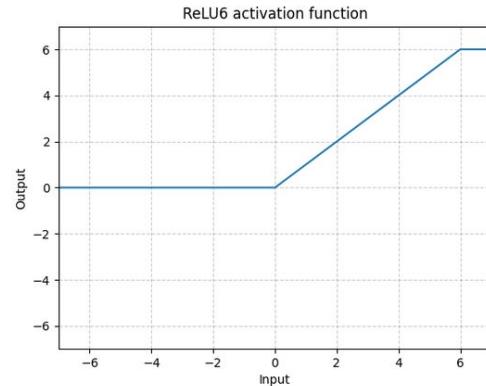
Activation functions for ‘underpowered’ platforms

A plethora of studies on implementing efficient DL models for platforms of low computational power and/or limited memory.

- Also: mapping DL models to fixed-point or even integer arithmetic.

Interesting ideas, for instance:

- The Relu6 activation function, designed to facilitate fixed-point implementation
 - Allows for better precision closer to 0.



MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, Hartwig Adam

<https://arxiv.org/abs/1704.04861v1>

Squeeze and Excite Blocks

Squeeze-and-Excitation Networks

Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu

<https://arxiv.org/abs/1709.01507>

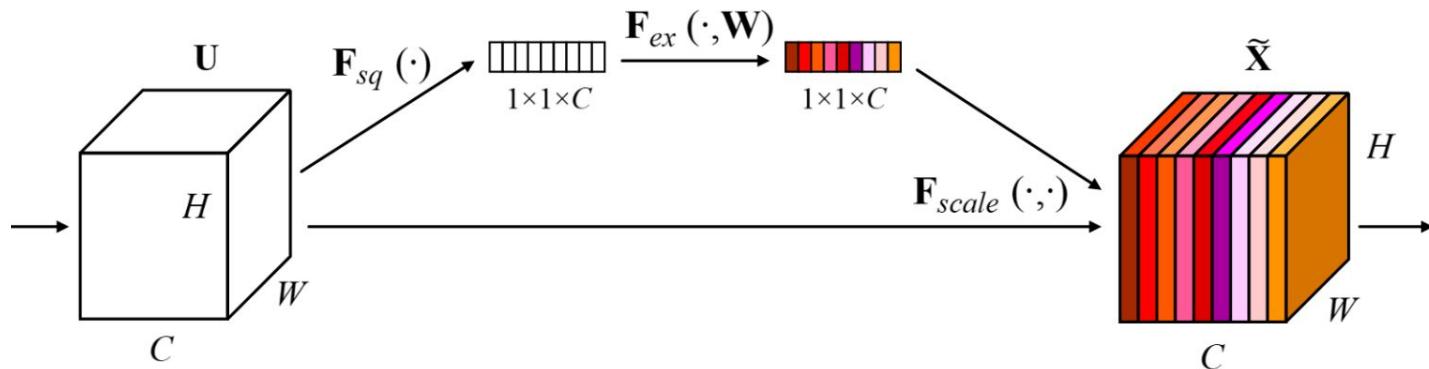
Squeeze and Excite Blocks: Background

- Application area: ConvNets
- Motivation: “Our goal is to improve the representational power of a network by explicitly modelling the interdependencies between the channels of its convolutional features.”
- The means: allow the network to perform *feature recalibration*, i.e. “learn to use global information to selectively emphasise informative features and suppress less useful ones.”

The gist of the idea: a gating (~attention) mechanism working across channels, based on the global (image-wide) statistics of per-channel activations.

Squeeze and Excite Block

- F_{sq} : squeeze operation: global, image-wide average pooling per channel
- F_{ex} : excitation operation: a small MLP with a ‘bottleneck’ (see next slide)
- F_{scale} : image-wide reweighting of channels, independently for each pixel

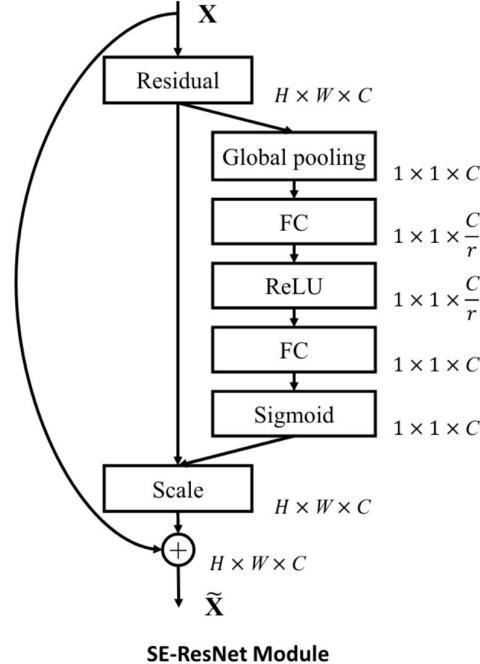
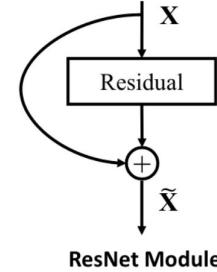
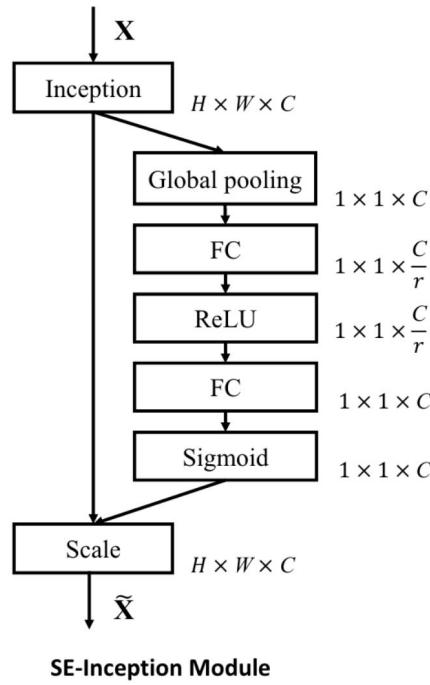
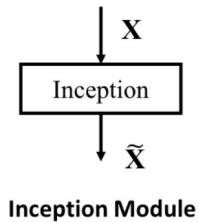


The MLP of excite operation

Two-layer MLP

- C inputs, C outputs
 - Note: a single-layer MLP would have $\sim C^2$ parameters, costly for large values of C !
- Layer 1: ReLU activation
 - Small number of units ($\ll C$, *reduction ratio r*) to improve generalization and reduce the number of parameters
 - Authors used $r = 8$
- Layer 2: Sigmoid activation
 - C units
 - Allows weighing each channel independently (in contrast to softmax, often used in gating mechanisms)
- The total number of parameters reduced to $O(C^2/r^2)$.
 - Useful recommendation to remember for other use cases!

Use cases (S&E-based modules)



Squeeze and Excite Blocks: Background

- Related: attention and gating mechanisms
- Can be plugged anywhere into ConvNets,
 - E.g. into UNets, <https://arxiv.org/abs/1904.08254> (uses S&E blocks everywhere)
- The authors won the first place in the ILSVRC 2017 classification competition (2.251% top-5 error, a ~25% relative improvement in comparison to the winner entry of the previous year).

Model optimization

Model distillation

Dataset distillation vs. knowledge distillation

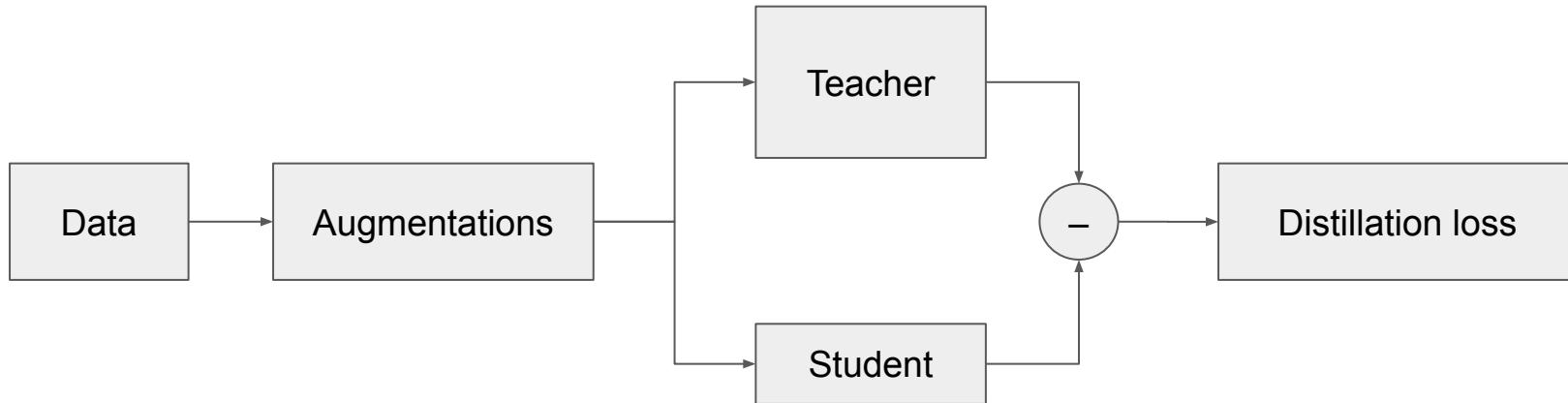
- Dataset distillation (not of interest in this section)
 - Preparation of the dataset, in particular filtering out undesirable observations/examples.
- Model distillation (special case of knowledge distillation):
 - Mapping a trained model into another (typically smaller/lighter) model.
Main types:
 - Response based distillation: minimizing the output differences between the teacher and student model. Uses distillation loss.
 - Feature-based distillation: distillation loss also defined at intermediate levels of processing (e.g. layers, correlation between outputs of layers).

Other line of division:

- offline distillation: pretrained, fixed teacher; only student subject to learning.
- online distillation: the student trained in parallel to the teacher.

Model distillation

- Advantages: can use unlabelled, augmented or artificial data for which the desired output is not known.
- Often results in good student models of quality equal or occasionally exceeding the teacher models



An example contribution to offline model distillation

Knowledge distillation: A good teacher is patient and consistent

Lucas Beyer, Xiaohua Zhai, Amélie Royer, Larisa Markeeva, Rohan Anil,
Alexander Kolesnikov

<https://arxiv.org/abs/2106.05237>

For a review, see also:

- Jianping Gou, Baosheng Yu, Stephen John Maybank, Dacheng Tao
Knowledge Distillation: A Survey, <http://arxiv.org/abs/2006.05525>

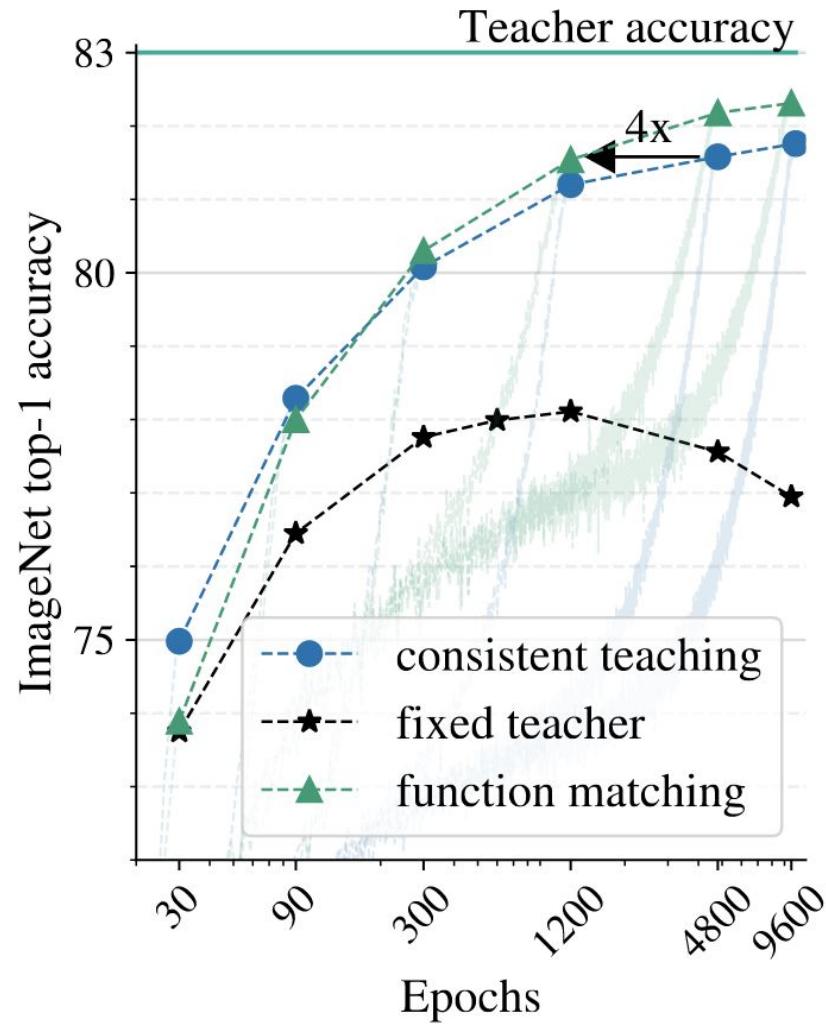
Main findings

1. Train long.
2. Do *not* initialize with teacher's weights.
 - a. May appear useful in shorter training runs, but ultimately underperforms.
3. Be consistent: Make sure teacher and student are queried on exactly the same piece of data (after augmentations).
4. Function matching: make the student mimic the teacher.
 - a. Use only the KL loss (or other teacher-student *relative* loss)
 - b. Do not use loss with respect to hard labels (targets).
5. If possible, make augmentations aggressive or interpolative (e.g. via mixup).
 - a. In KD, it may make sense to train the student on overly distorted inputs, as the goal here is *function matching*.

Results

The authors concentrate on:

- Teacher: BiT-M-R152x2 architecture:
a BiT-ResNet-152x2 (152 layers, ‘x2’
indicates the width multiplier)
pretrained on ImageNet-21k
- Student: BiT-ResNet-50
- Good generalization across different
image resolutions.
- The best published MobileNet v3
model.
- **A new state-of-the-art top-1
ImageNet accuracy of 82.82%.**



Mean Teachers

Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results

Antti Tarvainen, Harri Valpola

<http://arxiv.org/abs/1703.01780>

Key insights:

- Not a typical distillation: the teacher and the student learn in parallel.
 - In a sense, a variant of online distillation.
- More specifically, the teacher is ‘synthesized’ by ‘integrating’ over the history of recent students.
- Meant at obtaining a better model, not ‘shrinking’ it (because both models need to have ‘homomorphic’ architectures, with one-to-one correspondence of layers, units and weights).

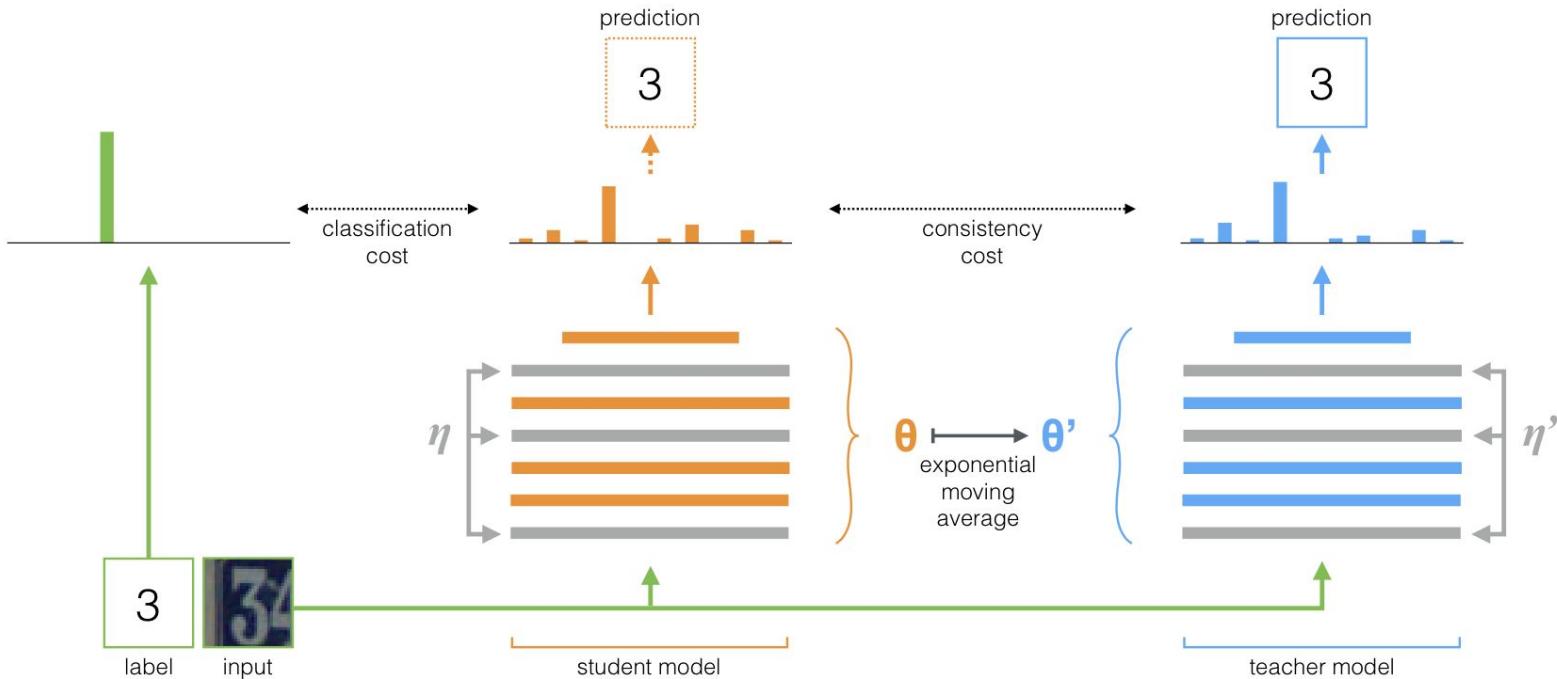


Figure 2: The Mean Teacher method. The figure depicts a training batch with a single labeled example. Both the student and the teacher model evaluate the input applying noise (η, η') within their computation. The softmax output of the student model is compared with the one-hot label using classification cost and with the teacher output using consistency cost. After the weights of the student model have been updated with gradient descent, the teacher model weights are updated as an exponential moving average of the student weights. Both model outputs can be used for prediction, but at the end of the training the teacher prediction is more likely to be correct. A training step with an unlabeled example would be similar, except no classification cost would be applied.

Loss function

Consistency loss:

$$J(\theta) = \mathbb{E}_{x, \eta', \eta} \left[\|f(x, \theta', \eta') - f(x, \theta, \eta)\|^2 \right]$$

where:

- θ and θ' are respectively the parameters of the student and the teacher,
- η and η' are independent noises applied to the student and the teacher.
 - Types of noise considered: Gaussian noise applied to inputs, dropout applied to the model, augmentations.

Notice: f is the same, i.e. the same model architecture.

Key idea: the teacher does not learn directly from data; the parameters of the teacher are estimated using the exponential moving average (EMA) of the students:

$$\theta'_t = \alpha \theta'_{t-1} + (1 - \alpha) \theta_t$$

Some results

Table 1: Error rate percentage on SVHN over 10 runs (4 runs when using all labels). We use exponential moving average weights in the evaluation of all our models. All the methods use a similar 13-layer ConvNet architecture. See Table 5 in the Appendix for results without input augmentation.

	250 labels 73257 images	500 labels 73257 images	1000 labels 73257 images	73257 labels 73257 images
GAN [25]		18.44 ± 4.8	8.11 ± 1.3	
Π model [13]		6.65 ± 0.53	4.82 ± 0.17	2.54 ± 0.04
Temporal Ensembling [13]		5.12 ± 0.13	4.42 ± 0.16	2.74 ± 0.06
VAT+EntMin [16]			3.86	
Supervised-only	27.77 ± 3.18	16.88 ± 1.30	12.32 ± 0.95	2.75 ± 0.10
Π model	9.69 ± 0.92	6.83 ± 0.66	4.95 ± 0.26	2.50 ± 0.07
Mean Teacher	4.35 ± 0.50	4.18 ± 0.27	3.95 ± 0.19	2.50 ± 0.05

Digression: an example of data distillation technique

Online Hard Example Mining <https://arxiv.org/abs/1604.03540>

- Common scenario: large number of easy examples and a small number of hard examples.
- Automatic selection of these hard examples can make training more effective and efficient.
- OHEM modifies SGD to sample from examples in a non-uniform way depending on the current loss of each example under consideration.
- The candidate examples are subsampled according to a distribution that favors diverse, high loss instances.

Model quantization

Model quantization

Analogous to distillation, but aimed at producing models that operate at lower computational precision.

The goals: reduction of

- primarily: computational cost of model querying,
- memory occupancy.

The means: ‘conversion’ of a model from its original floating-point form (typically 32-bit) to more compact domains, like:

- still floating-point: float16 (usually recommended for GPUs),
- integers: int16, int8 (good for CPUs).

Challenges: Limited precision, limited range, sometimes also asymmetric domain. ⁴⁰

Model quantization

Main types of quantization approaches:

- post-training quantization:
 - pure ‘conversion’ of model’s program/graph,
 - fast, simple,
 - good for third-party models and in scenarios where data is not available anymore.
 - Variants:
 - only weights (of units),
 - weights and activations (may require access to small representative dataset).
- quantization-aware training (QAT):
 - requires data and additional model training/tuning (time),
 - can engage unlabelled data!
 - will typically lead to better results,
 - in general preferred to post-training quantization.

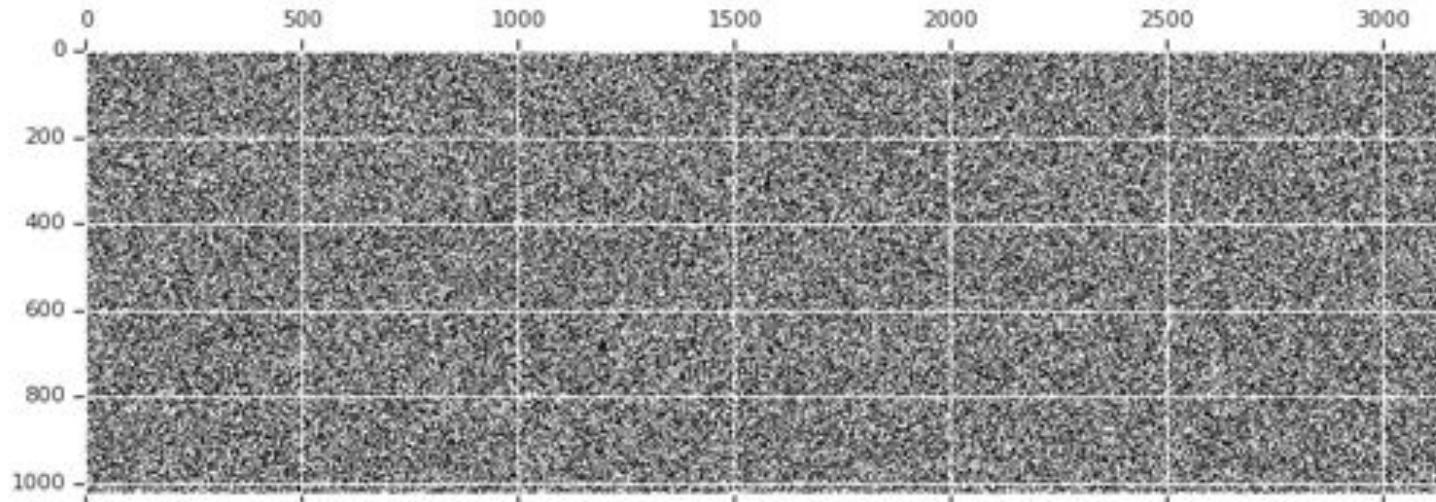
Model optimization

Model quantization techniques are part of a broader spectrum of model optimization techniques. Other approaches:

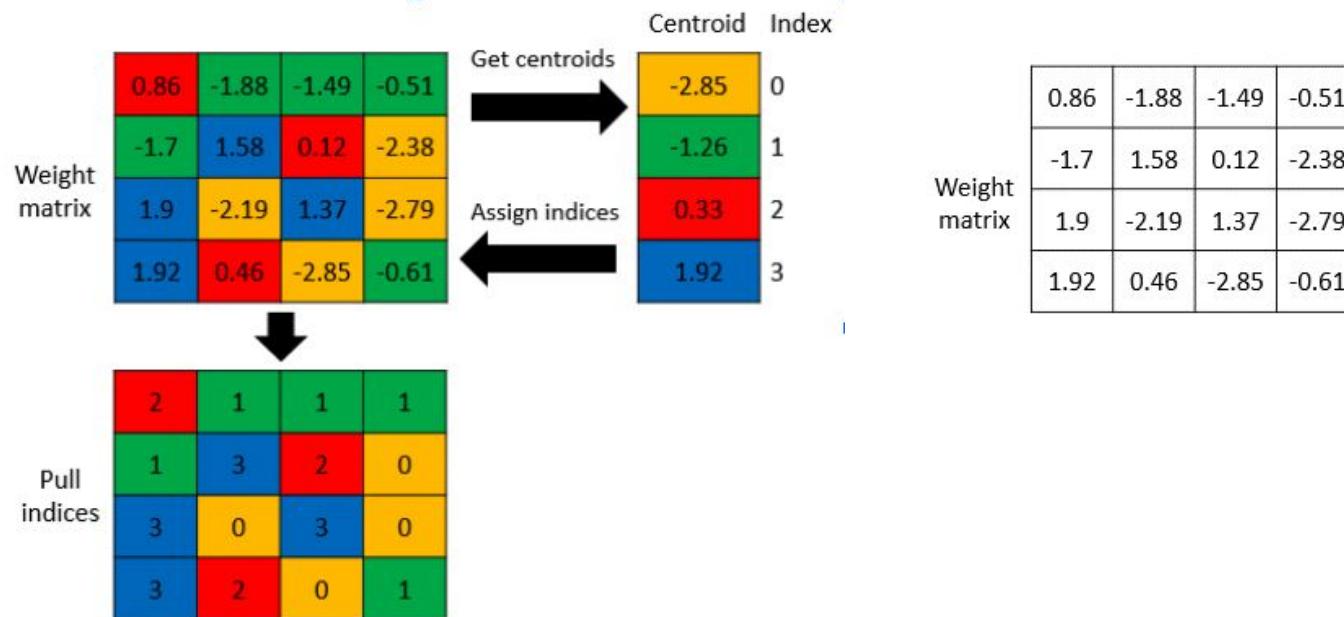
- Model pruning: removal of weights (and sometimes consequently entire units) that have marginal impact on the behavior of the model.
 - Typically requires sparse tensors: ‘tensor sparsification’
 - Can be done (preferably) during training.
- Weight clustering: grouping similarly-valued weights into clusters and index them with cluster centroids.
 - Meant mainly for reduction of disk/file occupancy.
 - However, some (quite exotic) hardware platforms offer accelerated inference for models with clustered weights.

Model pruning: An example

- Animation of a weight matrix pruning in consecutive epochs.
- Non-white pixels correspond to active (unpruned) weights.



Weight clustering: Illustration



More materials: https://www.tensorflow.org/model_optimization

Stochastic Weight Averaging (SWA)

Averaging Weights Leads to Wider Optima and Better Generalization

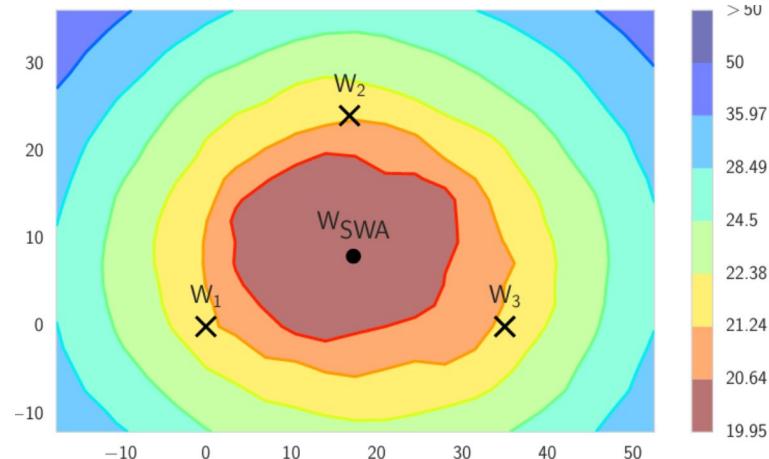
Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, Andrew Gordon Wilson

<https://arxiv.org/abs/1803.05407v3>

- Relatively popular and effective method of improving model's generalization
 - Essentially yet another technique of model optimization.
- Simple to implement
- Small to moderate computational overhead (compared to 'default' training)

Motivating observations

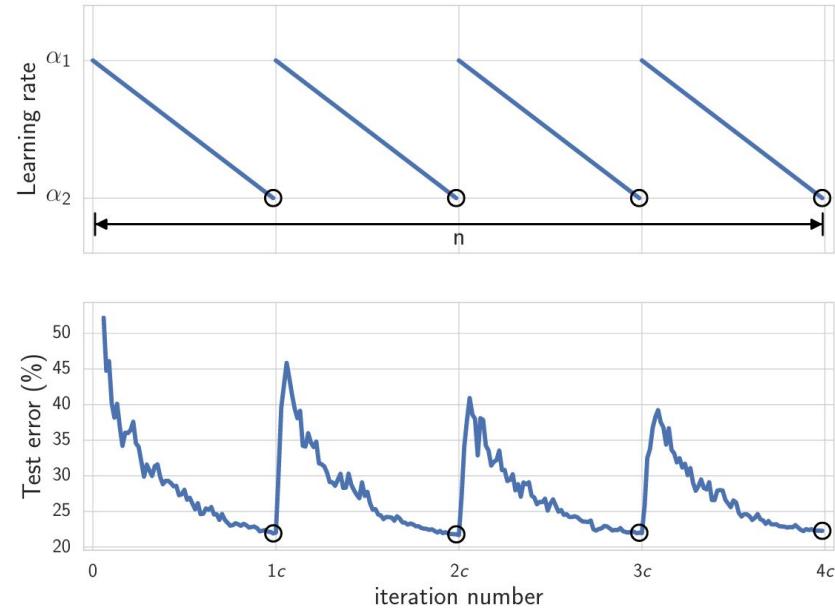
- Local optima found by SGD can be connected by simple curves of near constant loss.
- Training of a model quite often stops at the border of the plateau of local optimum.
 - For instance: three hypothetical weight configurations of three independent runs:
- The idea: average the weights of such networks, to obtain a more robust and better-generalizing model
- Challenges:
 - How to avoid the extra cost of training multiple models?
 - How to make averaging possible? (one cannot average completely independent models, as their weights do not correspond to each other)



Stochastic Weight Averaging (SWA)

Core idea: Cyclical learning rates

1. Train a model until reaching minimal assumed learning rate.
2. Save the model.
3. Increase the learning rate.
4. Repeat steps 1-3 a number of times.
5. Average the resulting models.



Implications:

- A single training run.
- Subsequent models are 'descendants' of the previous ones, so they are likely to have corresponding weights (weight averaging makes sense).

Cyclical learning rates in SWA

- Intuitive explanation: rapid increase of learning rate ‘kicks off’ the model, moving it to another configuration (yet still not too far from the same basin of error surface), from which it can arrive at the basin ‘from another angle’.
- The averaged weights are likely to be closer to the center of the basin.

The authors experimented with cyclical (left) and constant learning rates (right)

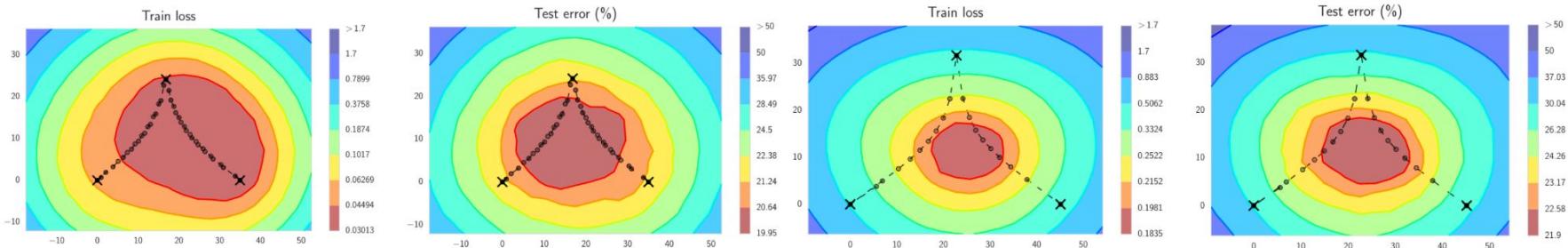
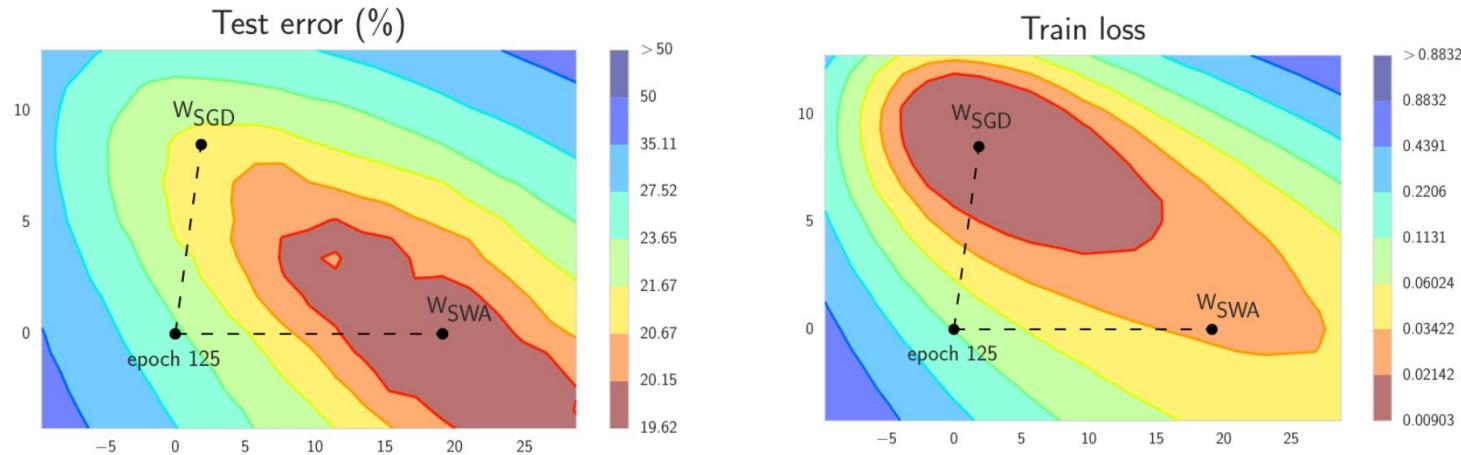


Figure 3: The L_2 -regularized cross-entropy train loss and test error surfaces of a Preactivation ResNet-164 on CIFAR-100 in the plane containing the first, middle and last points (indicated by black crosses) in the trajectories with **(left two)** cyclical and **(right two)** constant learning rate schedules. 9

Comparison of final weight configurations

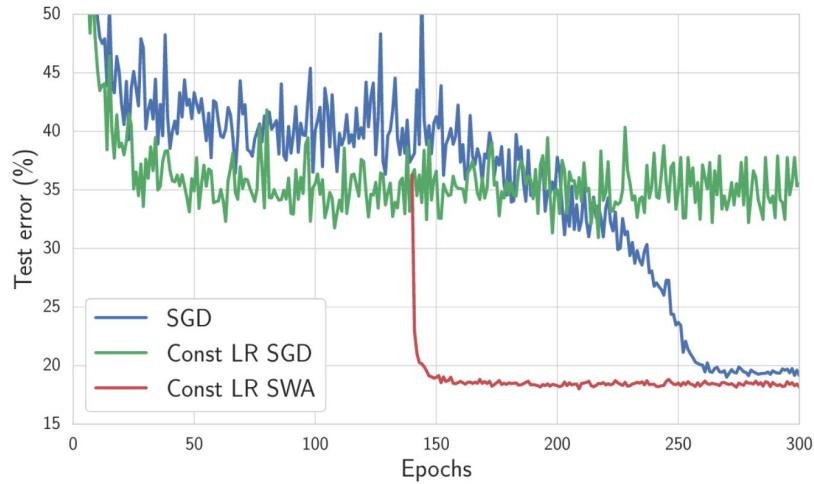


Test error and train loss surfaces showing the weights proposed by SGD (at convergence) and SWA, starting from the same initialization of SGD after 125 training epochs (Preactivation ResNet-164 on CIFAR-100).

SWA

Table 2: Top-1 accuracies (%) on ImageNet for SWA and SGD with different architectures.

DNN	SGD	SWA	
		5 epochs	10 epochs
ResNet-50	76.15	76.83 ± 0.01	76.97 ± 0.05
ResNet-152	78.31	78.82 ± 0.01	78.94 ± 0.07
DenseNet-161	77.65	78.26 ± 0.09	78.44 ± 0.06



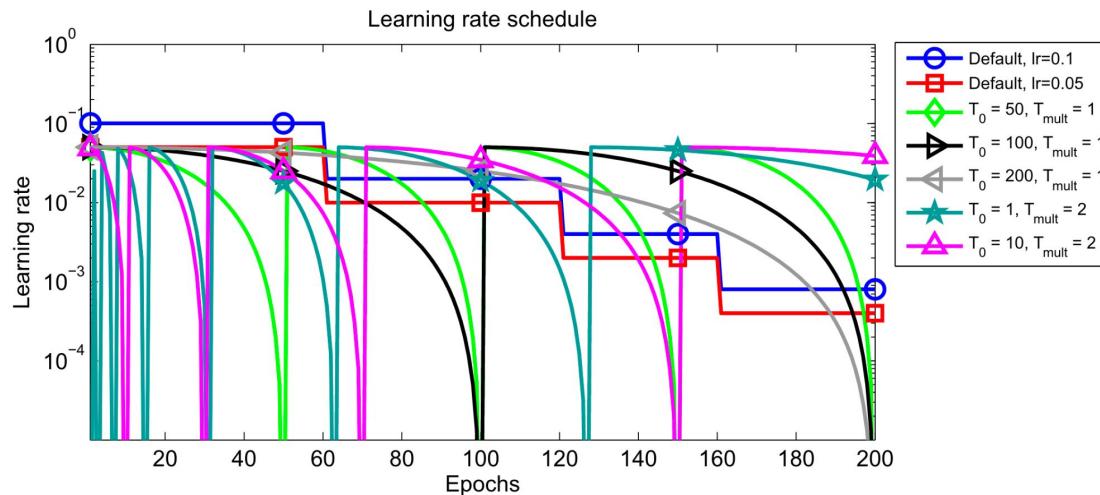
<https://github.com/timgaripov/swa>

Similar idea of *training schedules*: cosine annealing

SGDR: Stochastic Gradient Descent with Warm Restarts

Ilya Loshchilov, Frank Hutter

<https://arxiv.org/abs/1608.03983>



Lottery ticket hypothesis

Lottery Ticket Hypothesis

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks

Jonathan Frankle, Michael Carbin

<https://arxiv.org/abs/1803.03635>

An interesting approach that can be seen as a form of model optimization.

Motivation

- Neural network are heavy in terms of required computing power and memory.
- Pruning techniques can reduce the parameter counts of trained networks by over 90% without compromising accuracy.
- However, the sparse architectures produced by pruning are difficult to train from the start, which would similarly improve training performance.

Contributions

- Determining that a standard pruning technique uncovers subnetworks whose initializations made them capable of training effectively.
- Lottery ticket hypothesis: dense, randomly-initialized, feed-forward networks contain subnetworks (**winning tickets**) that—when trained in isolation—reach test accuracy comparable to the original network in a similar number of iterations.
 - The winning tickets have won the initialization lottery: their connections have initial weights that make training particularly effective.
- An algorithm to identify winning tickets and a series of experiments that support the lottery ticket hypothesis and the importance of ‘fortuitous initializations’.
 - Allows identifying winning tickets that are less than 10-20% of the size of several fully-connected and convolutional feed-forward architectures for MNIST and CIFAR10.

Practical implications

Winning tickets can be trained in isolation!

The approach:

1. Randomly initialize and train a network.
2. Prune the network.
3. Reset the remaining connections to their original initializations.
4. Train the pruned network.

Remarks:

- Uses a simple layer-wise pruning heuristic: remove a percentage ($p\%$) of the weights with the lowest magnitudes within each layer (Han et al. (2015))
- The authors used multiple iterations of the above schemata: n rounds of pruning, each round prunes $p^{1/n}\%$ of the weights.
- When the pruned network is reinitialized, this ceases to work.
 - Ergo, the initial values of parameters matter.

Experimental results

Comparison of ‘weight efficiency’ of winning tickets (solid lines) and randomly sampled subnetworks (dashed lines). See in particular the two insets on the right:

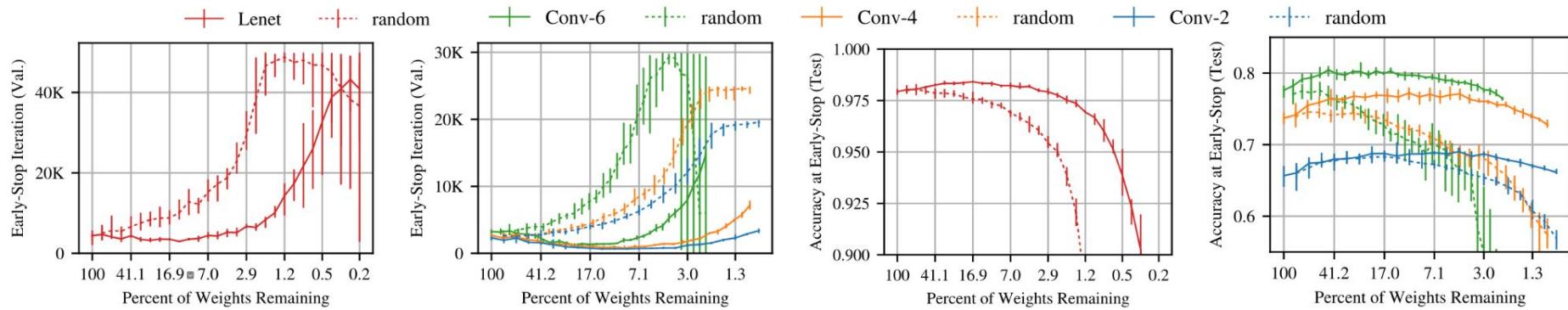


Figure 1: The iteration at which early-stopping would occur (left) and the test accuracy at that iteration (right) of the Lenet architecture for MNIST and the Conv-2, Conv-4, and Conv-6 architectures for CIFAR10 (see Figure 2) when trained starting at various sizes. Dashed lines are randomly sampled sparse networks (average of ten trials). Solid lines are winning tickets (average of five trials).

Other implications

- More efficient learning (see previous slide).
 - “sheds light on a new opportunity to dramatically reduce the cost of training the extraordinary models that are increasingly out of reach for all but the best resourced companies” *
- Help designing new, better architectures.
- Facilitate theoretical understanding of the networks.
- “(...) it surprisingly suggests that the capacity necessary for a neural network to learn a function is similar to the capacity necessary to represent it.” *

* CSAIL Talk: Friday 12-09-2022 The Lottery Ticket Hypothesis: On Sparse, Trainable Neural Networks

Hyperparameter optimization

Motivation:

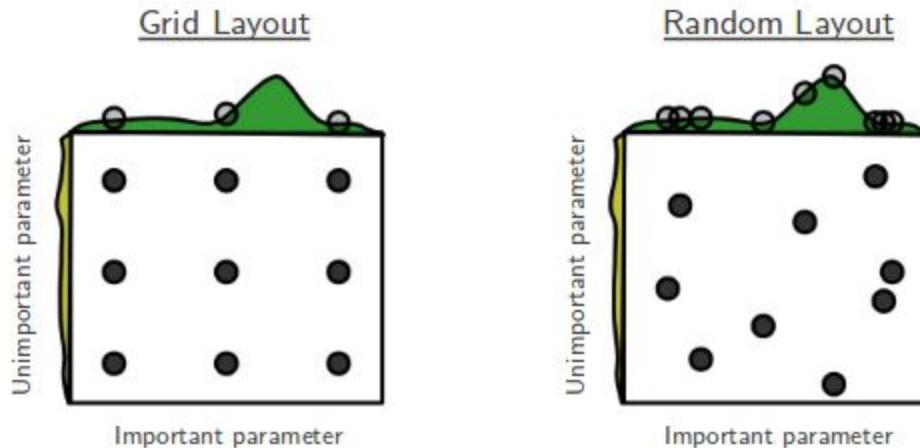
- There are many hyperparameters that SGD-based algorithms cannot optimize, including:
 - model architecture: e.g. number of layers, number of units per layer, number of channels in a convolutional layer, ...
 - training process: learning rate/schedule, parameters of the loss function (e.g. weights of regularization terms), early stopping conditions, ...
- We typically set them manually, based on experience, prior art, or ‘educated guess’
- Can this be automated? Sure!

Hyperparameter optimization

- Typical setup:
 - Input: list of hyperparameters (with domains) and the goal function/objective.
 - Goal: determining well-performing hyperparameter configurations for [Machine Learning] algorithms, datasets and applications at hand.
- But beware:
 - Combinatorial optimization (mixed types of domains of parameters)
 - This is black-box optimization, via trial-and-error: pick the combination of hyperparameter values, query the function => potentially computationally expensive.

Hyperparameter optimization

Advantages of random sampling compared to systematic sampling (grid search)



Hyperparameter optimization frameworks

SMAC

- One of the leading frameworks for HO.
- Uses Bayesian Optimization
- <https://github.com/automl/SMAC3>



SMAC

OptTuna

- <https://optuna.org/>



Both frameworks can be used to tune arbitrary ‘pipelines’ – they are not specific for Deep Learning.

Other components and techniques

MixUp augmentation

Mixup: Beyond Empirical Risk Minimization

Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, David Lopez-Paz

<https://arxiv.org/abs/1710.09412>

Motivations

- Neural networks trained with empirical risk minimization (ERM) change their predictions drastically when evaluated on examples just outside the training distribution (Szegedy et al., 2014), e.g. adversarial examples.
- Conventionally, one resort to data augmentation
 - (which has been formalized as Vicinal Risk Minimization (Chapelle et al., 2000) – requires defining a neighborhood around training examples)
 - Requires (some degree of) domain knowledge.
 - Assumes that the dependent variables (e.g. class assignment) remain unchanged under augmentations.

MixUp: Data-agnostic augmentation approach.

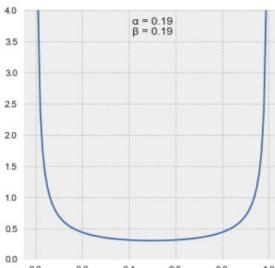
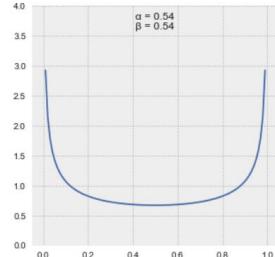
Given a pair of training examples (x_i, y_i) and (x_j, y_j) , mixup creates a virtual training example comprising of input and output:

$$\begin{aligned}\tilde{x} &= \lambda x_i + (1 - \lambda)x_j \\ \tilde{y} &= \lambda y_i + (1 - \lambda)y_j,\end{aligned}$$

where $\lambda \in [0, 1]$ is drawn from the beta distribution $\lambda \sim \text{Beta}(\alpha, \alpha)$
(suggested range of $\alpha \in [0.1, 0.4]$)

Main novelty/contribution:

- Augmentation affects also outputs, not only inputs.
- “Proportionality”: the same λ used in both spaces.



MixUp: Experimental results

Dataset	Model	ERM	<i>mixup</i>
CIFAR-10	PreAct ResNet-18	5.6	4.2
	WideResNet-28-10	3.8	2.7
	DenseNet-BC-190	3.7	2.7
CIFAR-100	PreAct ResNet-18	25.6	21.1
	WideResNet-28-10	19.4	17.5
	DenseNet-BC-190	19.0	16.8

(a) Test errors for the CIFAR experiments.

Dataset	ERM	<i>mixup</i>
Abalone	74.0	73.6
Arcene	57.6	48.0
Arrhythmia	56.6	46.3

Dataset	ERM	<i>mixup</i>
Htru2	2.0	2.0
Iris	21.3	17.3
Phishing	16.3	15.2

Table 4: ERM and *mixup* classification errors on the UCI datasets.

The authors examine the approach also on:

- Speech data
- Robustness to adversarial examples

Notice: MixUp is essentially ‘just’ a data augmentation technique.

Approximating large FC layers with truncated SVD

Motivation: Large dense layers are computationally heavy.

Idea: Replace the (trained) $u \times v$ weight matrix W with:

$$W \approx U \Sigma_t V^T$$

where:

- U : $u \times t$ matrix of first t left-singular vectors of W
- Σ_t : $t \times t$ diagonal matrix of the top t singular values of W
- V : $v \times t$ matrix comprising first t right-singular vectors of W .

$$\begin{array}{cccc} M_{m \times n} & = & U_{m \times m} & \Sigma_{m \times n} & V^*_{n \times n} \\ \begin{matrix} \text{gray grid} \\ \text{m rows} \\ \text{n columns} \end{matrix} & & \begin{matrix} \text{green} \\ \text{blue} \\ \text{m rows} \end{matrix} & \begin{matrix} \text{orange} \\ \text{yellow} \\ \text{m rows} \\ \text{n columns} \end{matrix} & \begin{matrix} \text{purple} \\ \text{m rows} \\ \text{n columns} \end{matrix} \\ U_{m \times m} & U^*_{m \times m} & \Sigma_{m \times n} & V^*_{n \times n} & \\ \begin{matrix} \text{green} \\ \text{blue} \\ \text{m rows} \end{matrix} & \begin{matrix} \text{green} \\ \text{blue} \\ \text{m rows} \end{matrix} & \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} & \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix} & \\ U & U^* & \Sigma & V^* & I_m \\ V_{v \times t} & V^*_{v \times t} & & & I_n \end{array}$$

(full SVD)

This reduces the number of parameters from uv to $t(u+v)$.

Technical realization: two fully connected layers (no nonlinearity between them!):

- The first layer uses $\Sigma_t V^T$ as weight matrix (no biases)
- The second layer used U as weight matrix (and biases taken from W).

Module 10:

Automatic differentiation

Krzysztof Krawiec

Wydział Informatyki i Telekomunikacji
Politechnika Poznańska
2024

<http://www.cs.put.poznan.pl/kkrawiec/>



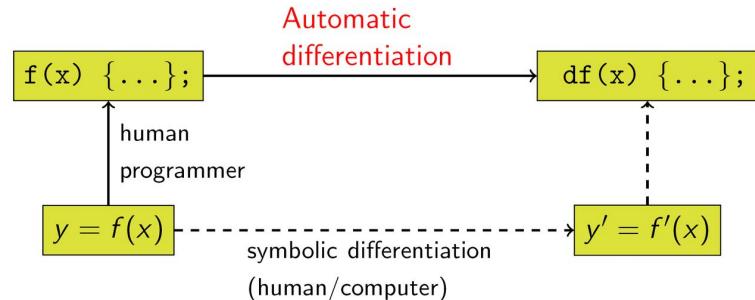
Automatic differentiation

Automatic differentiation

Automatic differentiation (AD, autodiff) is a repertoire of techniques for efficient calculation of derivatives of computer programs (here: DL models).

Main difference w.r.t. symbolic differentiation:

- applied to programs, not formulas.



Most AD techniques rely on program transformations:

- given a program, generate a program that calculates its derivative,
- transformation is typically assumed to be conducted once (e.g. at compile time, or when loading the model and readying it for execution).

Automatic differentiation

The task of autodiffing a complex program can be decomposed in a number of ways.

Important division:

- forward-mode AD
- reverse-mode AD

Most ‘compatible’ with error backpropagation, and hence most common in deep learning.

Autodiff: an example

Based on the blog of Tom Ellis: <http://h2.jaquarpaw.co.uk/posts/automatic-differentiation-worked-examples/>

Assume we need to calculate the derivative of the following program:

```
p = 7 * x
r = 1 / y
q = p * x * 5
v = 2 * p * q + 3 * r
```

Inputs: x, y

Output: v

Required preparatory steps before autodiff:

1. Use prefix functions instead of operators
2. Removal of nested subexpressions
(resulting from step 1)

Transformed program
prepared for AD:

```
p = mul (7, x)
r = div (1, y)
i1 = mul (p, x)
q = mul (i1, 5)
i2 = mul (2, p)
i3 = mul (i2, q)
i4 = mul (3, r)
v = add (i3, i4)
```

Forward-mode AD

The forward mode transformation applies the appropriate differentiation rule to each line in the input program (left) to obtain a derivative line (right).

- Each line processed independently.
- One rule per line.
- More precisely: the variables (dp , dr , ...) store the infinitesimal quantities.

```
p = mul (7, x) | dp = mul (7, dx)
r = div (1, y) | dr = negate (div (dy, mul (y, y)))
i1 = mul (p, x) | di1 = add (mul (dp, x), mul (p, dx))
q = mul (i1, 5) | dq = mul (di1, 5)
i2 = mul (2, p) | di2 = mul (2, dp)
i3 = mul (i2, q) | di3 = add (mul (di2, q), mul (i2, dq))
i4 = mul (3, r) | di4 = mul (3, dr)
v = add (i3, i4) | dv = add (di3, di4)
```

Forward-mode AD

Properties/advantages:

- Uses space (memory) proportional to the original program,
- The runtime is proportional to the runtime of the original program.
- A `d_` variable can be discarded as soon as the `_` variable is not needed anymore.

Reverse-mode AD

Requires further preprocessing of the program: explicit variable duplication, so that each variable is used only once

- (notice the relation to the *linear logic* and *linear type systems*).

```
p = mul (7, x)      (x1, x2) = dup x
                      p = mul (7, x1)
r = div (1, y)       (p1, p2) = dup p
i1 = mul (p, x)     r = div (1, y)
q = mul (i1, 5)     i1 = mul (p1, x2)
i2 = mul (2, p)     q = mul (i1, 5)
i3 = mul (i2, q)     i2 = mul (2, p2)
i4 = mul (3, r)     i3 = mul (i2, q)
v = add (i3, i4)    i4 = mul (3, r)
                     v = add (i3, i4)
```

We need this transformation because *usages* in the original program will become *variable definitions* in the reverse mode program (and vice versa).

Reverse-mode AD

Generating the code (`d_dx` is meant to express d/dx , the differentiation operator in the Leibniz notation):

```
(x1, x2) = dup x | d_dx = add (d_x1, d_dx2)
p = mul (7, x1) | (_, d_dx) = mul (d_dp, (x1, 7))
(p1, p2) = dup p | d_dp = add (d_dp1, d_dp2)
r = div (1, y) | d_dy = negate (div (d_dr, mul (y, y)))
i1 = mul (p1, x2) | (d_dp1, d_dx1) = mul (d_di1, (x, p1))
q = mul (i1, 5) | (d_di1, _) = mul (d_dq, (5, di1))
i2 = mul (2, p2) | (_, d_dp2) = mul (d_di2, (p2, 2))
i3 = mul (i2, q) | (d_di2, d_dq) = mul (d_di3, (q, i2))
i4 = mul (3, r) | (_, d_dr) = mul (d_di4, (r, 3))
v = add (i3, i4) | (d_di3, d_di4) = dup (d_dv)
```

The final program is formed by concatenating:

- the program (left) with
- the sequence of derivative calculations (right) in reverse order.

Note: `d_dv` is assumed to ‘arrive’ from outside of the program.

Reverse-mode AD

Properties:

- The runtime is proportional to the runtime of the original program.
- Uses space (memory) proportional to the runtime of the original program,

For instance x_1 needs to be kept around almost until the very end of the execution of the AD program (recall the latter is executed bottom-up):

```
(x1, x2) = dup x | d_dx = add (d_x1, d_dx2)
p = mul (7, x1) | (_, d_dx) = mul (d_dp, (x1, 7))
(p1, p2) = dup p | d_dp = add (d_dp1, d_dp2)
r = div (1, y) | d_dy = negate (div (d_dr, mul (y, y)))
i1 = mul (p1, x2) | (d_dp1, d_dx1) = mul (d_di1, (x, p1))
q = mul (i1, 5) | (d_di1, _) = mul (d_dq, (5, di1))
i2 = mul (2, p2) | (_, d_dp2) = mul (d_di2, (p2, 2))
i3 = mul (i2, q) | (d_di2, d_dq) = mul (d_di3, (q, i2))
i4 = mul (3, r) | (_, d_dr) = mul (d_di4, (r, 3))
v = add (i3, i4) | (d_di3, d_di4) = dup(d_dv)
```

Why forward and reverse mode?

At first sight, the forward mode seems more useful – it has both time and space complexity proportional to that of the original program.

- Things get more complex once we take into account the sizes of the data structures being differentiated (numbers of variables, or vector/array sizes).

In general, for a program $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$

- the forward mode requires fewer operations when $m > n$,
- the reverse mode requires fewer operations when $m < n$.

In practice, $m < n$ is more common (programs are ‘contracting’ the data).

- However, one can hybridize these modes (change the ordering of multiplications of the Jacobian matrices that store the derivatives).

$$\underbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}_{|\mathbf{y}| \times |\mathbf{x}|} = \underbrace{\frac{\partial h(\mathbf{b})}{\partial \mathbf{b}}}_{|\mathbf{y}| \times |\mathbf{b}|} \underbrace{\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}}}_{|\mathbf{b}| \times |\mathbf{a}|} \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{|\mathbf{a}| \times |\mathbf{x}|}$$

See nice explanation in terms of sizes of Jacobian matrices:

<https://math.stackexchange.com/questions/2195377/reverse-mode-differentiation-vs-forward-mode-differentiation-where-are-the-be>

Further considerations

Related/further topics:

- How to calculate derivatives of loops and conditionals?

A lot of ongoing research and R&D work.

- Often part of sophisticated graph optimization algorithms implemented by framework-specific tools.
 - E.g. Grappler in TensorFlow.
 - Perform also other optimization: materialization of constants, elimination of common subexpressions, layout optimization (e.g. NHWC vs. NCHW), cost of copying the data between CPU and GPU, function inlining, and more.
 - For instance, TF profiler reveals functions that combine *both* the dot product and apply an activation function.
 - Take into account the technical specifics of the target platform: CPU vs. GPU, available memory, detects dead code, etc.

The algebra behind autodiff: dual numbers

Let's extend the domain of real numbers to *dual numbers*.

$$z = a + b\epsilon$$

where $a, b \in \mathbb{R}$, and ϵ has the following property: $\epsilon^2 = 0$ (*nilpotence*). Note the analogy to extending the reals \mathbb{R} to complex numbers \mathbb{C} .

Consider $f(x) = x^2$. Now, what is $f(x)$ if x is a dual number?

$$f(a + b\epsilon) = (a + b\epsilon)^2 = a^2 + 2ab\epsilon + b^2\epsilon^2 = f(a) + 2ab\epsilon = f(a) + f'(a)b\epsilon$$

The algebra behind autodiff: dual numbers

Is this a coincidence?

No. For any polynomial:

$$P(a + b\epsilon) = P(a) + bP'(a)\epsilon$$

Even more generally, this holds for any analytic function, by the virtue of Taylor expansion:

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)b^n\varepsilon^n}{n!} = f(a) + bf'(a)\varepsilon$$

(note what happens to Taylor expansion terms for $n \geq 2$)

Practical upshot: dual numbers allow us to calculate gradients (symbolically) alongside with the signals propagated through an ANN.