Fantastic neural networks and how to train them

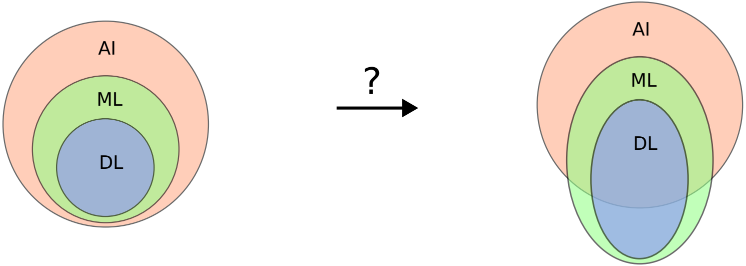
Nikolai Hartmann

LMU Munich

April 24, 2024, LMU Joint Particle Physics Group Seminar

Buzzword definitions for this talk

The great thing about buzzwords - i can choose my own definition



- **Machine Learning:** Fitting, but we don't really care what exactly the model is (in classical fitting we usually have an interpretation for the parameters)
- **Deep learning:** Solving problems with neural networks i can't solve with BDTs (usually involving larger datasets and multiple layers)
- **A**rtificial **I**ntelligence: Emulating human intelligence/behavior (i want to draw a blurry boundary to the stuff we never did by hand before)

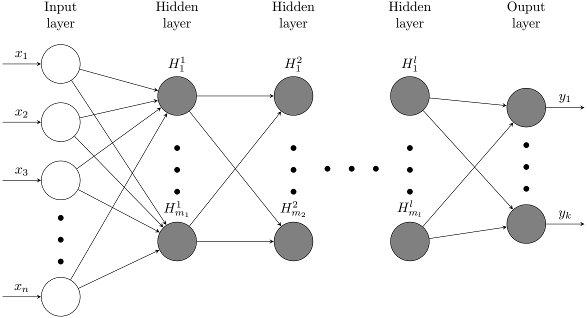
Supervised learning

Focussing on supervised learning in this talk
$\rightarrow$ visit talk by David Gisegh (26.06.) to learn more about unsupervised learning!

- Want to find a function that maps a set of input features $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ to a set of output features $\mathbf{y} = [y_1, y_2, \ldots, y_n]$
- We only have (typically simulated) **training examples**
- Want to find (multidimensional) parametrisation of something that we can only simulate $\rightarrow$ **inverse problem**
- Two main goals:
  - **Classification:** map inputs to labels $y_i \in 0, 1$ (or a probability $p_i \in [0, 1]$)
  - **Regression:** predict continuous values $y_i \in \mathbb{R}$

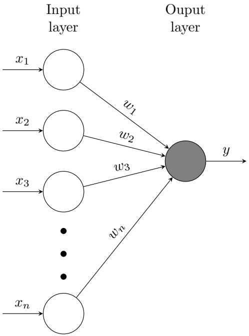
The fully-connected, feedforward neural network

Aka multilayer perceptron (MLP)

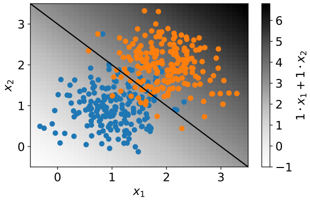


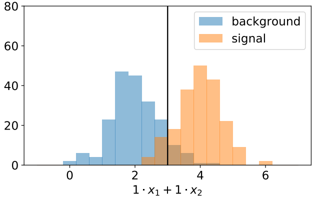
- Propagate information through the network by taking the weighted sum of inputs at each neuron and applying an activation function $\sigma(\sum_i w_i x_i + b)$
- Activation function adds non-linearity
  $\rightarrow$ can approximate any function with sufficient number of neurons!
- Each connection corresponds to one weight w
- Each neuron has one bias b
- Classification: one output neuron per possible label

The simplemost “neural network”



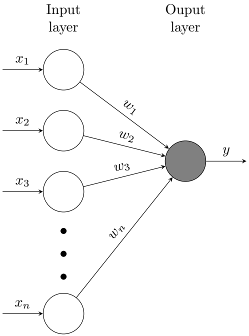
Example: 2D $w_1 = w_2 = 1$



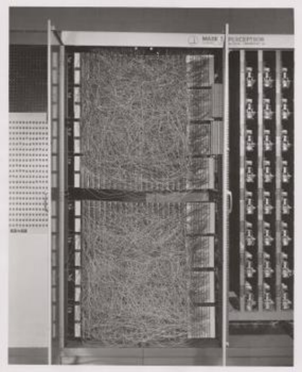


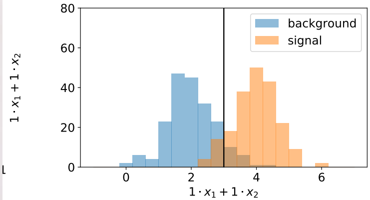
- A single neuron (no hidden layer) corresponds to linear discriminant
  $\rightarrow$ Output $= \sum w_i x_i$
- Idea goes back to 1957 - the “Perceptron” (in Hardware!) by Frank Rosenblatt

The simplemost “neural network”



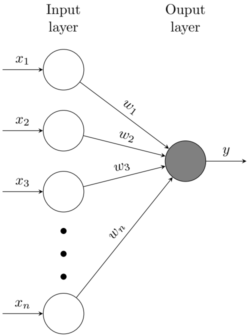
Example: 2D $w_1 = w_2 = 1$



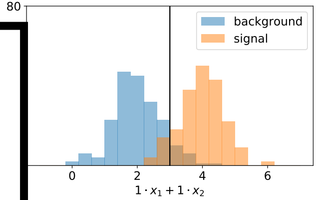


- A single neuron (no hidden la … liscriminant
  $\rightarrow$ Output $= \sum w_i x_i$
- Idea goes back to 1957 - the … !) by Frank Rosenblatt

The simplemost "neural network"



Example: 2D $w_1 = w_2 = 1$



New York Times, July 7, 1958:

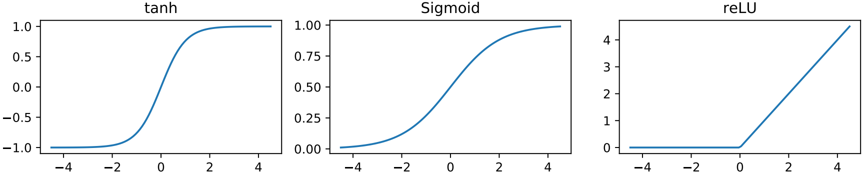
"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, reproduce itself and be conscious of its existence."

- A single neuron (no hi ... nant
  $\rightarrow$ Output $= \sum w_i x_i$
- Idea goes back to 1957 - the "Perceptron" (in Hardware!) by Frank Rosenblatt

The power of hidden layers

- Hidden layers without activation functions don't help
  $\rightarrow$ linear combination of linear combinations is still a linear combination
- Non-linear activation function at each neuron in the hidden layer(s) allows to approximate **any** function! (given enough neurons)
  $\rightarrow$ proven for sigmoid in 1989 by George Cybenko
  $\rightarrow$ more generally proven in 1991 by Kurt Hornik
- One hidden layer is in principle enough
- Experience: multi-hidden-layer networks work better
  $\rightarrow$ "**Deep neural networks**"

Activation functions



- Derivatives:

tanh:

$$f'(x) = 1 - f(x)^2$$

sigmoid:

$$f'(x) = f(x)(1 - f(x))$$

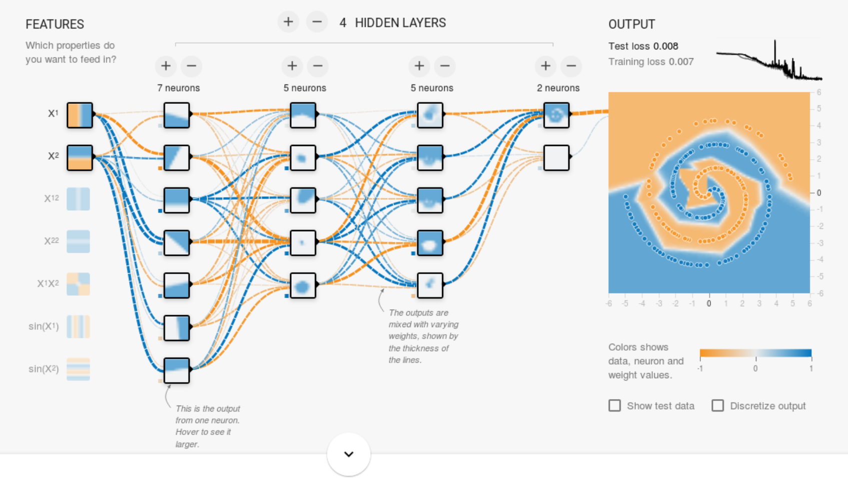
reLU:

$$f'(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$$

- tanh function in the hidden layers was popular for a long time
- Problem: gradient vanishes for large input values
  $\rightarrow$ especially problematic in multi-layer networks
- most popular nowadays: reLU and variants of it
- sigmoid is used wherever output should be in $[0, 1]$

Example

https://playground.tensorflow.org



Loss function

To solve the optimisation problem we need a measure for the distance between the current (pred) output and the desired (true) output

Mean squared error (MSE):

$L_{\mathrm{MSE}} = \frac{1}{N} \sum_i (y_i^{\mathrm{pred}} - y_i^{\mathrm{true}})^2$

$\rightarrow$ good for Regression, mean absolute error also popular

Cross entropy (CE):

$L_{\mathrm{CE}} = -\sum_{i=1}^{N_{\mathrm{classes}}} y_i \log \hat{y}_i$

$\rightarrow$ good for Classification, same as **maximum Likelihood**

Binary cross entropy (BCE) - for 2 classes, 1 output:

$L_{\mathrm{BCE}} = -\frac{1}{N} \sum_i \left[y_i^{\mathrm{true}} \ln y_i^{\mathrm{pred}} + (1 - y_i^{\mathrm{true}}) \ln(1 - y_i^{\mathrm{pred}}) \right]$

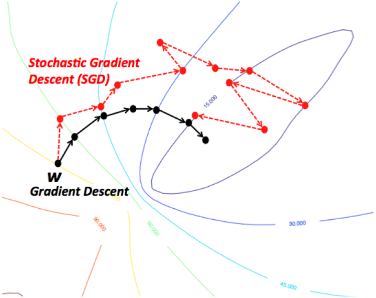
Backpropagation

The algorithm that makes neural networks work

- Know all operations and their derivatives the computation graphs
  $\rightarrow$ can use **chain rule**
  $\rightarrow$ compute once **forward**, store intermediate values, then **backward** to get gradient
- Single variable: $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$
- Multivariable: $\mathbf{J}_f(g(x)) = \mathbf{J}_f(g)\mathbf{J}_g(x)$, in components $\frac{\partial f_i}{\partial x_j} = \frac{\partial f_i}{\partial g_k} \frac{\partial g_k}{\partial x_j}$
- For derivative of **scalar** (loss): $\frac{\partial f}{\partial x_j} = \underbrace{\frac{\partial f}{\partial g_k}}_{\text{vector}} \underbrace{\frac{\partial g_k}{\partial x_j}}_{\text{Jacobian}}$
  $\rightarrow$ matrix multiply gradient (row) **vector** with the **Jacobian** in each step
  $\rightarrow$ referred to as vector-Jacobian-product (**VJP**)
- The cool thing: Usually not required to fully compute the Jacobian to get the VJP! (e.g. a single matrix multiplication to get VJP for matrix output w.r.t matrix input)

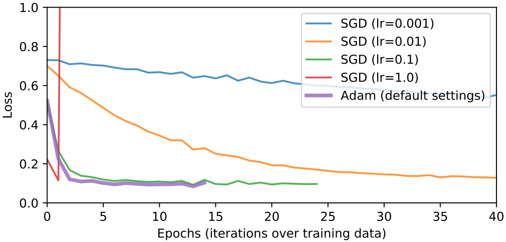
Stochastic Gradient Descent (SGD)

- Loss function is usually averaged over all training examples
  $L_{\text{tot}} = \frac{1}{n} \sum_i L_i$
- Need to propagate all training examples through the network for each gradient update
  $\rightarrow$ computationally intense for large training sets



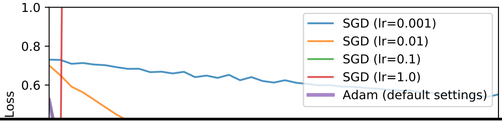
- Solution: Gradient updates on random subsets ("batches") of training data
- batch size gives a handle for tradeoff:
  number of gradient steps $\leftrightarrow$ iterations over dataset ("**epochs**")

Extensions to SGD



- Need to adjust the step size (“**learning rate**”) for good convergence
- Many approaches
  - schedule learning rate during training (start high, decrease, warmup, cosine schedule, ...)
  - use information on previous changes (“**momentum**”)
  - do this parameter wise
  - use second order moments of the gradients
  - ...
- Lots of research happening - keep an eye on it!
- Current (~~2018~~ 2024) best default choice: “**Adam**” and variants of it
  $\rightarrow$ works very well in default settings in most cases

Extensions to SGD



- Need to adjust the … te
- Many approaches
  - schedule learning rate during training (start high, decrease, warmup, cosine schedule, ...)
  - use information on previous changes ("**momentum**")
  - do this parameter wise
  - use second order moments of the gradients
  - ...
- Lots of research happening - keep an eye on it!
- Current (~~2018~~ 2024) best default choice: "**Adam**" and variants of it
  $\rightarrow$ works very well in default settings in most cases

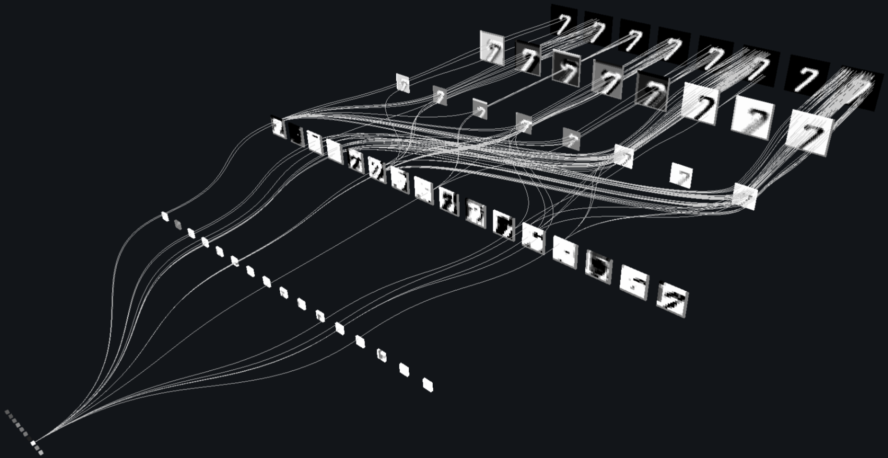
NN Architectures

Most architectures make use of symmetries in the data

- Translation invariance: **Convolutional neural networks (CNNs)**
  $\rightarrow$ "slide" a neural network over neighboring inputs (e.g. pixels)
- Sequential data: **Recurrent neural networks (RNNs)**
  $\rightarrow$ Stateful neurons, feed output back in, together with input of next time step
- Permutation invariance (and/or equivariance):
  - Sets without predefined relations: **Deep Sets**
    $\rightarrow$ process each element individually
    $\rightarrow$ aggregate globally over hidden states in permutation invariant way (e.g. sum)
  - Graphs: **Graph (convolutional) networks (GNNs)**
    $\rightarrow$ aggregate over neighbors in graph
  - **Transformers**: can be seen as graph networks with fully connected graph
    $\rightarrow$ now also standard for sequential data (see LLMs)

Moving more and more towards the permutation invariant architectures
(both in AI research and HEP ML)

Convolutional neural networks (CNNs)

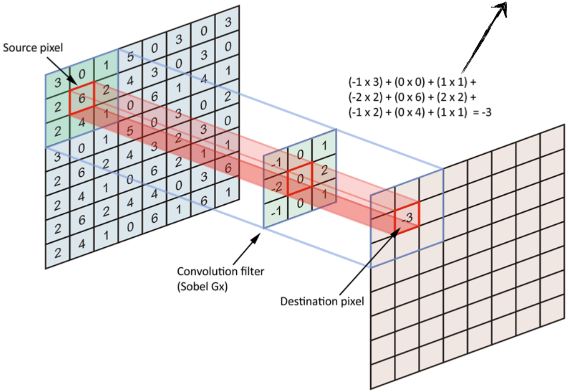


http://terencebroad.com/nnvis.html

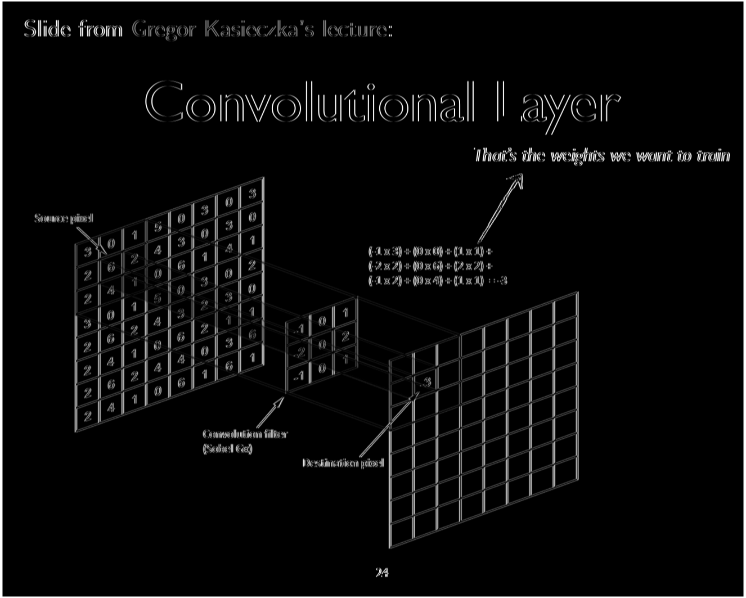
Slide from Gregor Kasieczka's lecture at Terascale ML school 2018

Convolutional Layer

That's the weights we want to train

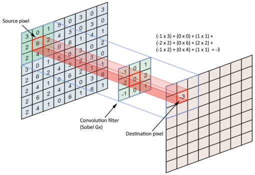


Slide from Gregor Kasieczka's lecture at Terascale ML school 2018

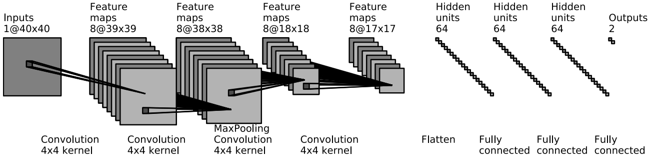


Slide from Gregor Kasieczka's lecture at Terascale ML school 2018:

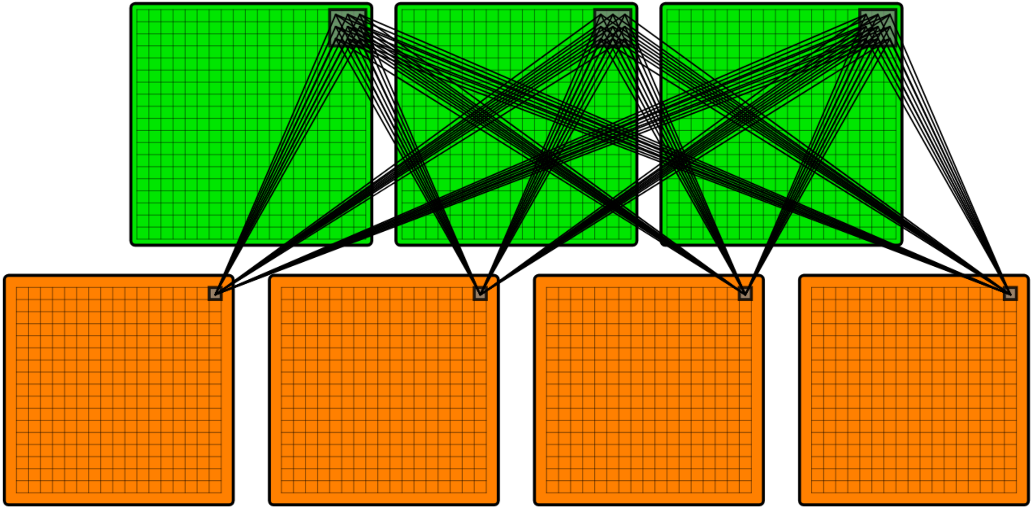
Convolutional Network



- ***How to build a convolutional network***
  - Chain multiple conv layers
  - Use multiple masks per layer
  - Pooling
    - Max Pooling
    - Average Pooling
  - Add a fully connected network in the end

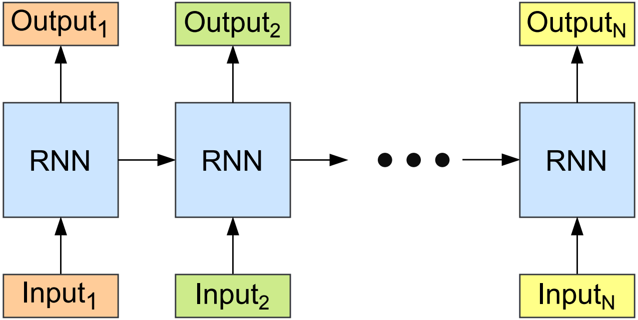


How $N_{\text{input}} \to M_{\text{output}}$ channels work:



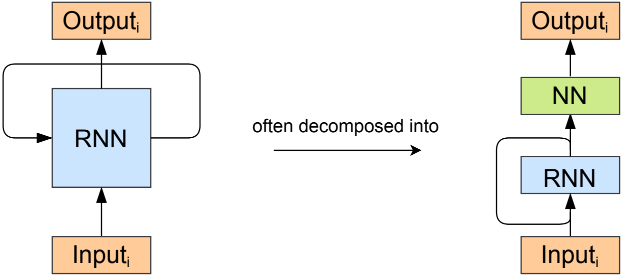
$\to$ animation

RNNs



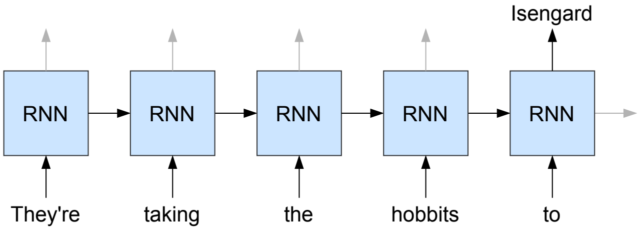
- Operate on a sequence, passing-on a hidden **state**
- **Shared weights** across the sequence
- Usually thought of as a sequence **in-time**, but can be any **ordered sequence**

RNNs



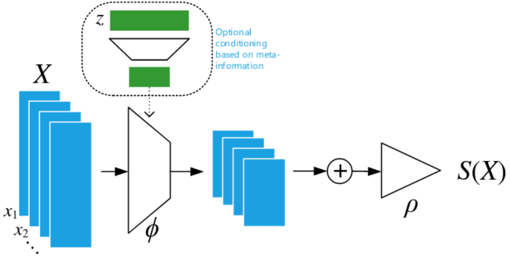
- Operate on a sequence, passing-on a hidden **state**
- **Shared weights** across the sequence
- Usually thought of as a sequence **in-time**, but can be any **ordered sequence**

RNNs



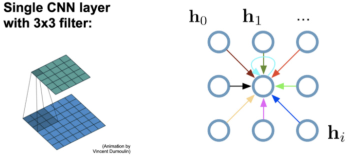
- Operate on a sequence, passing-on a hidden **state**
- **Shared weights** across the sequence
- Usually thought of as a sequence **in-time**, but can be any **ordered sequence**
- Used to be the standard for language models, but not anymore (Transformers took over)
  $\rightarrow$ also in particle physics it seems their time is mostly over ...

Deep Sets



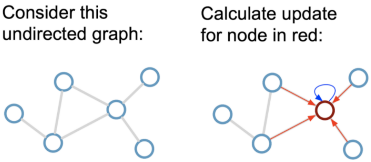
- **Per-item transformation** ϕ (e.g. MLP - shared weights!) followed by
- **Permutation invariant aggregation** (e.g. sum)
- Every **permutation-equivariant** ($f(\pi(x)) = \pi(f(x))$) transformation allowed for per-item step
  $\rightarrow$ e.g. add/concatenate global sum to each item
- Output is now fixed-length vector, can be transformed by another MLP
- Very simple to implement, give it a try!
  $\rightarrow$ Popularized in HEP by "Energy flow networks" paper (also soft/collinear safe variant)

Graph networks



Full update:

$$\mathbf{h}_4^{(l+1)} = \sigma\left(\mathbf{W}_0^{(l)}\mathbf{h}_0^{(l)} + \mathbf{W}_1^{(l)}\mathbf{h}_1^{(l)} + \cdots + \mathbf{W}_8^{(l)}\mathbf{h}_8^{(l)}\right)$$

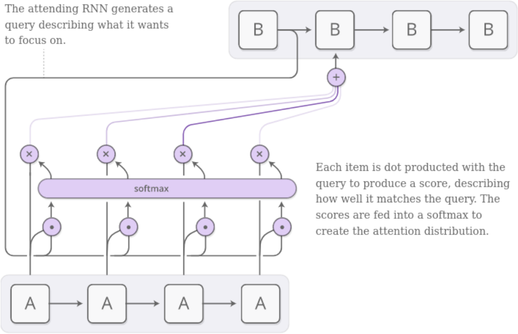


Update rule:

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}^{(l)}\right)$$

- Update node featuers by **sum over neighbors in graph**
  $\rightarrow$ similar to sum over neighboring pixels in CNN
- Can't have fixed weight (no meaningful ordering of neighbors, number not constant)
- Simplest option: sum without weights (Graph convolutional network, GCN)
- More advanced: work with features on edges, features of neigboring nodes (e.g. attention)
  - in general can pass information from nodes to edges, edges to nodes ...
  - ... and to and from global features

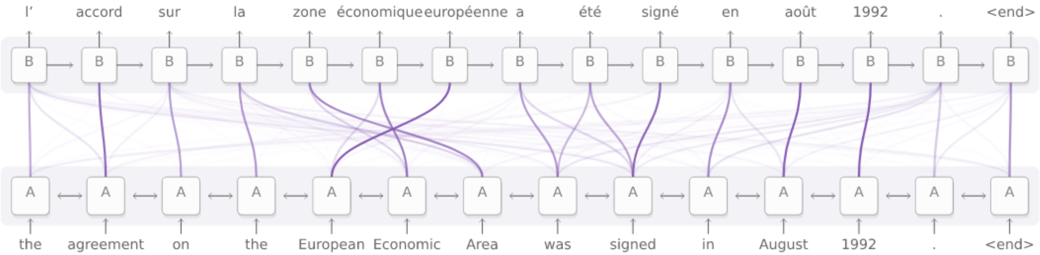
Attention



- Started as an attempt to improve translation tasks with RNNs
- Have each element of one sequence *attend* to elements of another sequence
- Possible implementation: score from dot product of each encoder, decoder step pair
- Precursor of transformers - *Attention is all you need*

[1] https://distill.pub/2016/augmented-rnns

Example for machine translation

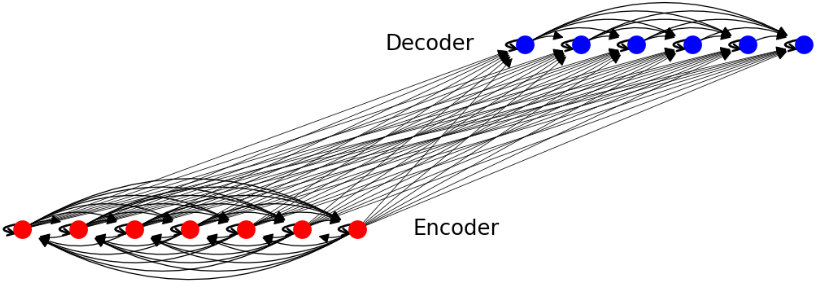


Input and target sequence can also be the same - **Self Attention**

[1] https://distill.pub/2016/augmented-rnns

Transformers as a Graph Network

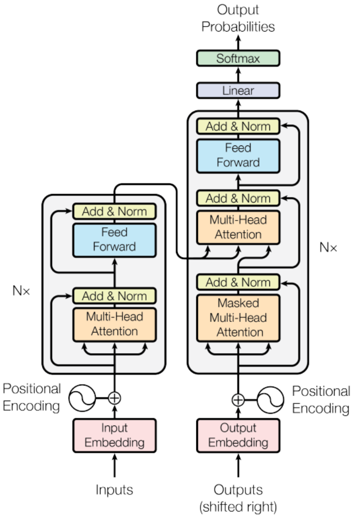
from https://docs.dgl.ai/en/latest/tutorials/models/4_old_wines/7_transformer.html



- Lines represent attention weights - inferred from features of nodes they connect
- **Decoder:** only connections to previous tokens (*causal mask*)
- **Encoder:** fully connected graph for attention
- Encoder-only: BERT, ParT
- Decoder-only: GPT(1,2,3)

Transformer details

Attention is all you need (2017) (arXiv:1706.03762)



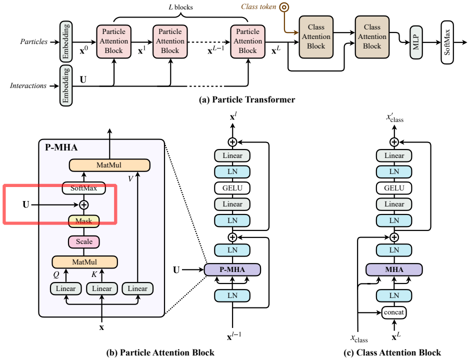
- Uses Multi-Head-Attention (MHA)
- MLP (with one hidden layer) after each MHA block
- Skip connections and normalization layers make deep models possible

Public data sets

- Public data sets help exchange and development of common models
- Nice example: **Top tagging dataset** (arXiv:1902.09914, 10.5281/zenodo.2603255)
  $\rightarrow$ Leading 200 jet constituents for $\approx$ 1M pythia (boosted) jets with Delphes detectors sim
  $\rightarrow$ Task: find out if the jet is normal QCD jet or comes from a top quark
  $\rightarrow$ Huge amount of architectures has been tested, often generally applicable

ParT

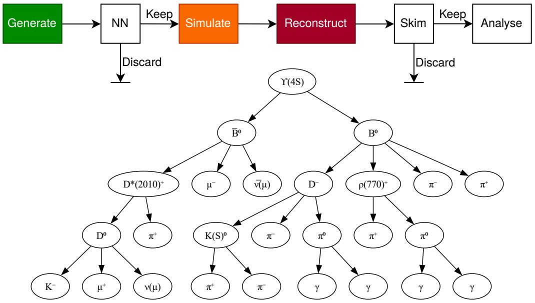
Particle Transformer for Jet Tagging arXiv:2202.03772



Modifications w.r.t. standard transformer:

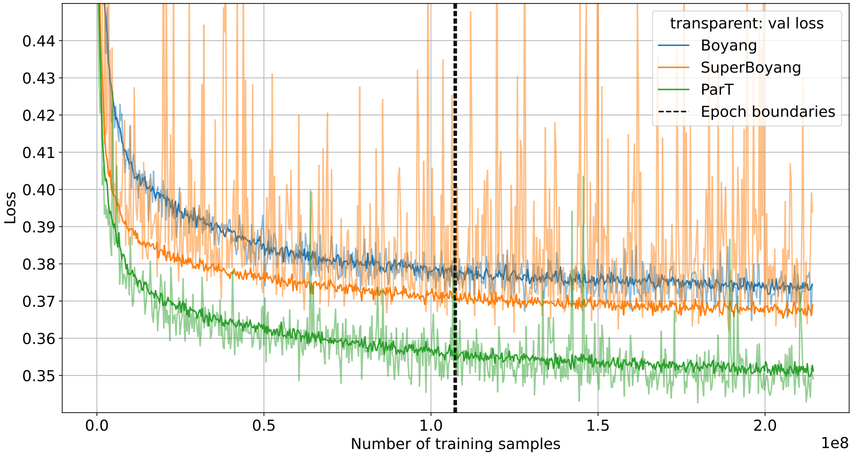
- Add embedded interaction features (e.g. invariant mass) as bias term to attention score $\rightarrow$ very similar to attention mechanism with edge features in graph networks
- Attention to class token to produce global classification result
- State-of-the-art for jet tagging if trained on large enough datasets (100M events)

Preliminary results on Belle II Smart Background project



- Use a NN as a MC filter
  $\rightarrow$ predict after event generation which events we will throw out later
- Graph neural networks, using the generator-level decay tree work well
- But maybe we have been fooled and it's mainly about the correlation between particles?
  $\rightarrow$ try ParT, can still feed in adjacency matrix as pair feature

Preliminary results on Belle II SmartBackground project



→ almost out-of-box better performance than our prevously optimized models!

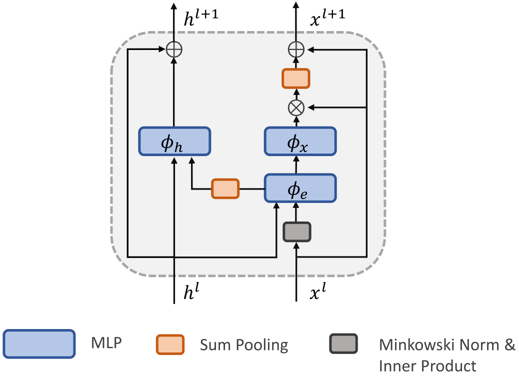
Working with Lorentz vectors

Working with 4-momentum vectors we can make use of Lorentz symmetry!

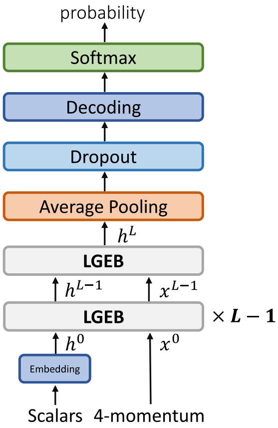
- Lorentz covariant quantities of a set of 4-momentum vectors can be constructed as functions of pairwise Minkowski inner products $f(p_1, p_2, ..., p_n) = f(\{p_i p_j\}_{i,j})$
- Two architectures with state-of-the art (2024) performance on jet-tagging tasks:
  - **LorentzNet** (arXiv:2201.08187): build a *Minkowski dot product attention* based on this
    $\rightarrow$ transform a set of 4-vectors into a new set of 4-vectors across layers
  - **PELICAN** (arXiv:2211.00454): run rank $2 \rightarrow 2$ permutation equivariant transformations
    $\rightarrow$ run transformation on the whole matrix of pairwise Minkowski products
    $\rightarrow$ needs fewer parameters than other models (but maybe more computation?)

LorentzNet

arXiv:2201.08187



Lorentz Group Equivariant Block (LGEB)



LorentzNet

$\rightarrow$ Caspar is applying this for background suppression in the $B \rightarrow K^* \nu\nu$ analysis!

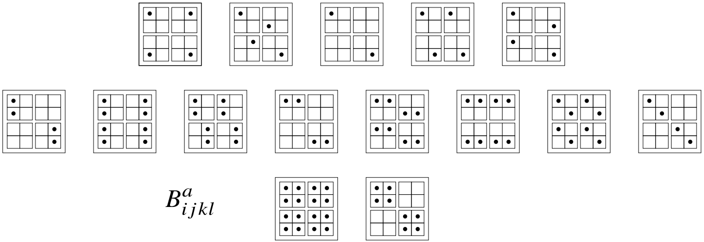
PELICAN

arXiv:2307.16506

Equivariant Layer: $T^{(\ell+1)} = \textsc{Agg} \circ \textsc{Msg}\left(T^{(\ell)}\right).$

$\textsc{Agg}$

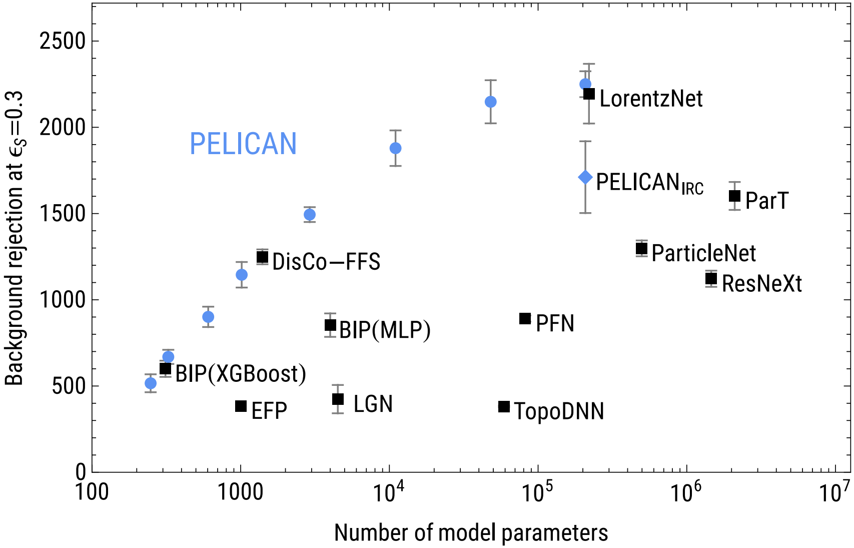
$$T'^{a}_{ij} = \sum_{k,l=1}^{N} B^{a}_{ijkl} T_{kl}.$$



B^{a}_{ijkl}

15 possible permutation equivariant matrix $\rightarrow$ matrix aggregators run in each layer

Performance on Top tagging



How to train on a large dataset

Preparing the data input pipline often requires a significant fraction of the work

Some recommendations:

- Do all preprocessing that doesn't blow up the amount of data much before
- Store in parquet or feather files (use pandas for flat tables, awkward array for the rest)
- Stuff that blows up data (padding with 0s, features for all pairs ...) better on-the-fly
- If data fits into memory, load it into memory
    - estimate memory usage before - one float takes 4 bytes
      $\rightarrow$ 8 GiB memory: 2.6M events, 100 particles each, 8 features per particle
    - in practice will need factor 2-3 more due to copies (not always easy to avoid)
- If not, load in large chunks (100k - 1M events per chunk)
    - typically 1 chunk = 1 file
    - generate randomized batches from the chunk
    - load random chunks
    - useful pattern: subprocess loads next chunk while training on previous chunk runs

Summary

- Neural networks most useful when we process lower level data
  → for high-level observables in tabular form, use BDTs! MLPs rarely beat them
- Processing lower level data usually requires making use of symmetries
  → CNNs, RNNs, Deep Sets, GNNs, Transformers
- Following developments on public datasets can be very useful
  → jet tagging is very active and the methods often generalizable
- Typical particle physics data format: List of 4-momentum vectors (+extra features)
  → ideal use case for Transformers (work well with small modifications, see ParT)
  → ... but also need large datasets (10-100M events)
- Smaller datasets can profit from imposing Lorentz symmetry
  → the most interesting approaches to date: LorentzNet, PELICAN
- Or having pre-trained models that can be fine-tuned to the samller datasets
  → ParT shows promising results for jet tagging
  → "Predict-the-next-particle" approaches getting more popular as well
  → e.g. arXiv:2305.10475, 2403.05618, 2401.13537

Outlook

- Let's talk more about the "boring" topic how we actually train these networks
  $\rightarrow$ usually large fraction of work in data preparation, setting up the input pipeline
  $\rightarrow$ we will likely see larger models and run our stuff on multiple GPUs soon
- Access to resources is not very uniform, for example here in Munich:
  - 3 local GPUs in the AG-Kuhr
  - 4 GPUs in C2PAP, need to apply for project - setup via VM (lrz cloud) or slurm queue
  - several multi-GPU nodes in lrz AI system, more to come
    $\rightarrow$ access can be requested by everyone with linux cluster account

If you want to learn more

- Machine learning schools by the ErUM-Data-Hub
  https://erumdatahub.de/veranstaltungen
- ODSL block courses, one currently going on (until today)
  https://indico.ph.tum.de/event/7606
  $\rightarrow$ slides and recordings online!
- Nice video tutorials by Andrej Karpathy (*Neural Networks: Zero to Hero*)
  https://www.youtube.com/playlist?list=PLAqhIrjkxbuWI23v9cThsA9GvCAUhRvKZ