Python for Data Science 2

Lecture 18 (2-3 Sessions)

Amir Farbin

Motivations

- ***Curse of Dimensionality***- Data occupies a small fraction of high dimensional space

- ***Manifold Learning***

    - Natural Data Lives in low dimensional (Non-Linear) Manifold.

    - For example consider 100 by 100 pixel images of faces

        - We can imagine that images a specific person's face trace a manifold in pixel space

            - as we rotate face wrt 3 angles

            - as facial change by movement of O(50) muscles on face

    - Ideally feature extractors would learn these manifolds.

(a) Azimuth (pose)

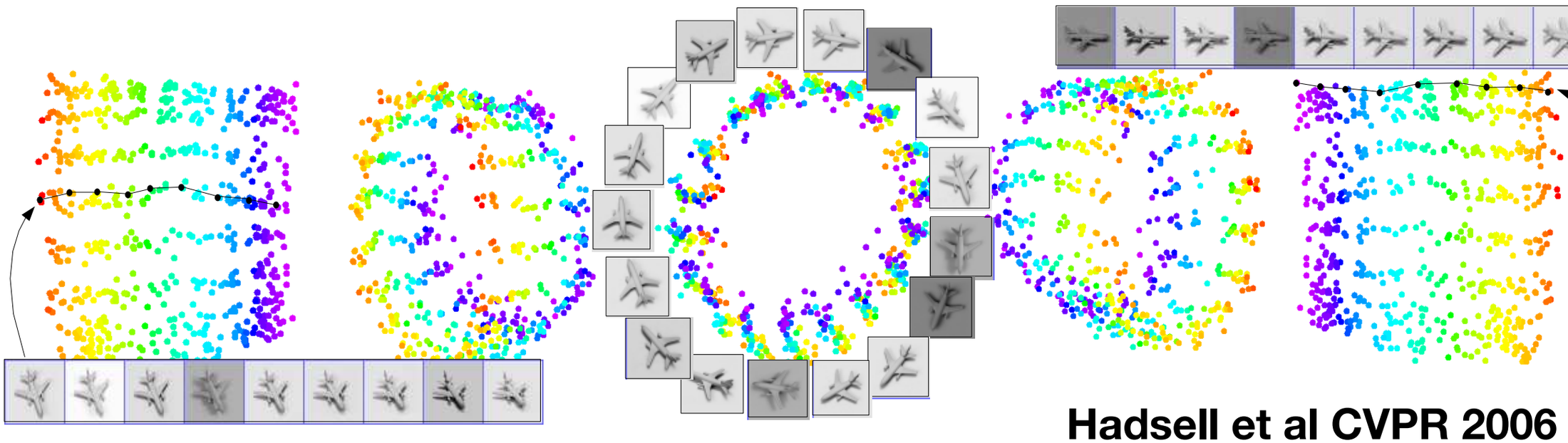
(b) Elevation

(c) Lighting

(d) Wide or Narrow

Figure 3: **Manipulating latent codes on 3D Faces:** We show the effect of the learned continuous latent factors on the outputs as their values vary from -1 to 1. In (a), we show that one of the continuous latent codes consistently captures the azimuth of the face across different shapes; in (b), the continuous code captures elevation; in (c), the continuous code captures the orientation of lighting; and finally in (d), the continuous code learns to interpolate between wide and narrow faces while preserving other visual features. For each factor, we present the representation that most resembles prior supervised results [7] out of 5 random runs to provide direct comparison.

https://arxiv.org/pdf/1606.03657.pdf



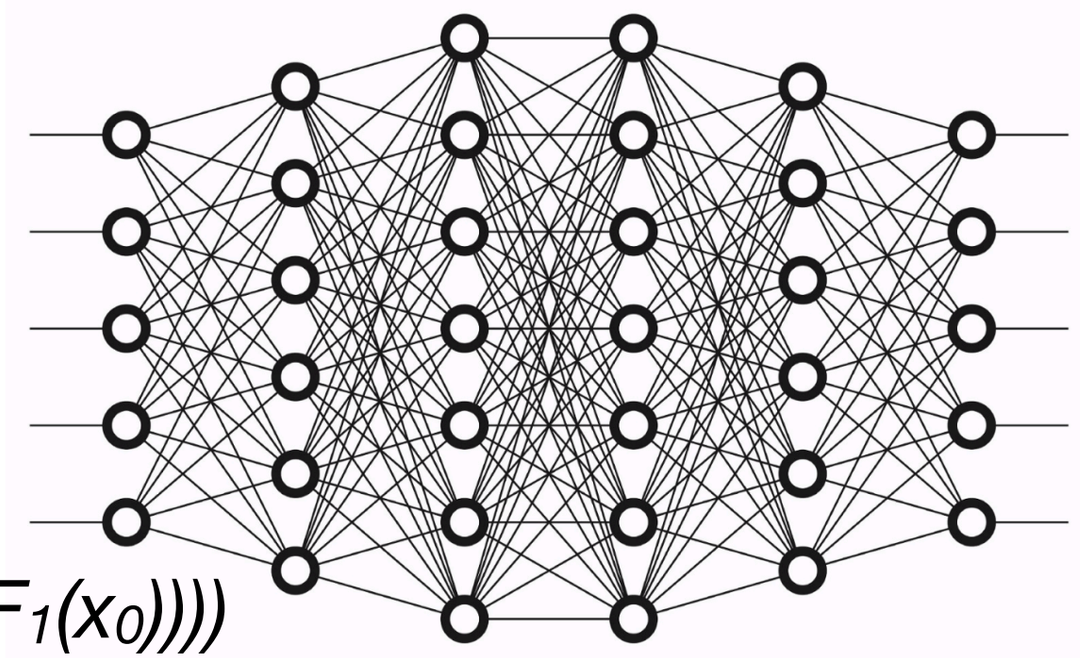
Hadsell et al CVPR 2006

Why Deep?

- "We can approximate any function as close as we want with shallow architecture. Why would we need deep ones?"
    - Deep machines are ***more efficient*** for representing certain classes of functions
    - They can represent more complex functions with ***less "hardware"***
- ***Hierarchy of representations*** with increasing level of abstraction.
    - ***Images***: Pixel→Edge→Texton→Motif→Part→Object
    - ***Text***: Character→Word→Word Group→Clause→Sentence→Story
    - ***Speech***: Samples→Spectral Band→Sound→…→Phone→Phoneme→Word
- In DL, these are learned features…
    - Each stage transforms input representation into high-level representation
    - High-level are more global/invariant
    - Low-level are shared among categories.

Building DNNs

Dense Networks



- Simplest type.
- Single layer: $F_i(x_{i-1}) = f(\mathbf{W}_i\, x_{i-1} + b_i)$
  - d deep network $\Longrightarrow F_d(F_{d-1}(F_{d-2}(\ldots(F_1(x_0))))$
    - Each $\mathbf{W}_i$ is n_i by n_{i-1} Matrix $\Longrightarrow$ n_i is width at depth i
- d and $\{n_i\}$ are hyper-parameters
- Note: thin/deep networks don't have enough bandwidth to propagate info. Generally want $n_i >> d$.

- https://playground.tensorflow.org/

Ingredients

Inputs

- Neural Networks work best with inputs ~ 0 $\longrightarrow$ 1

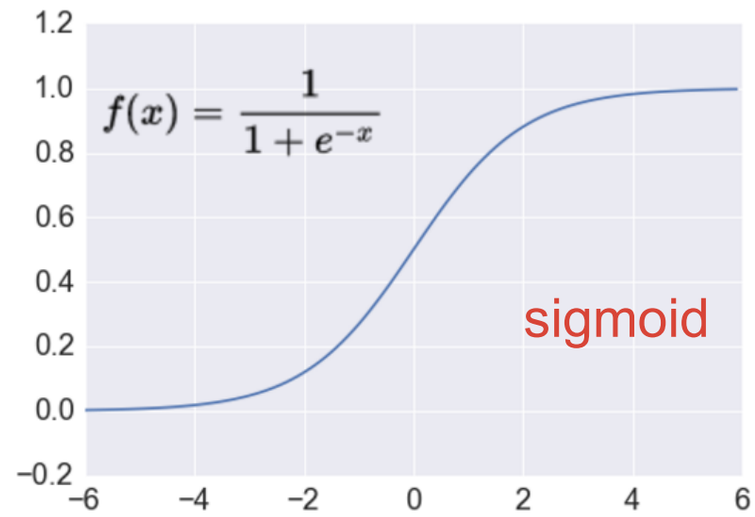
- You can shift by mean and divide by variance

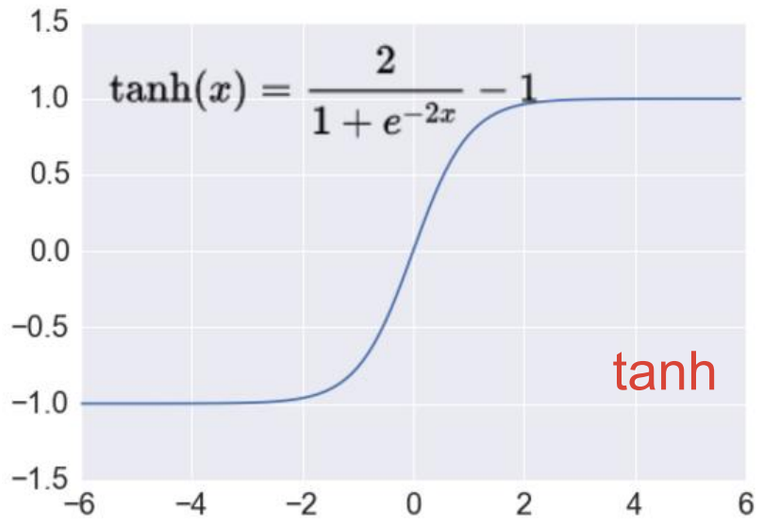
- Or use a scaler from sklearn.

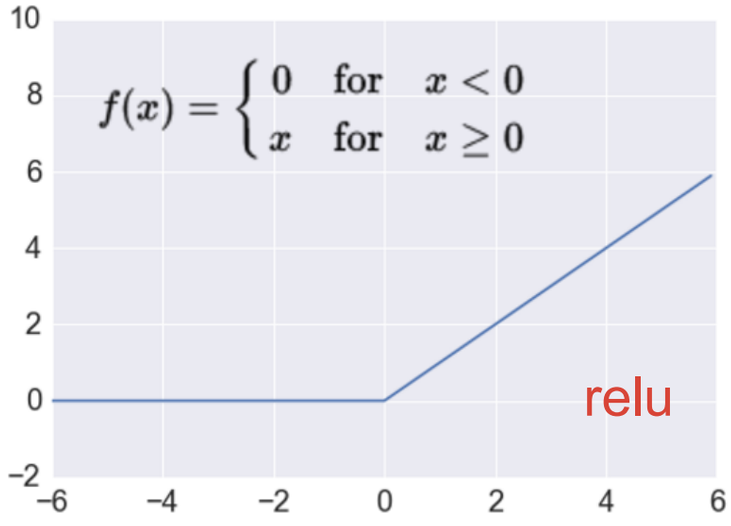
Activations

Activations

- Vanishing Gradients
- Sigmoid
  - Saturate
  - non-Zero centered
- Relu
  - Larger Gradient
  - Simple to compute
  - If learning rate too high, neurons can “Die” (never activate)
- Leaky Relu
- Use Linear as output of Regression tasks
- Choice of activation can be seen as a hyper parameter





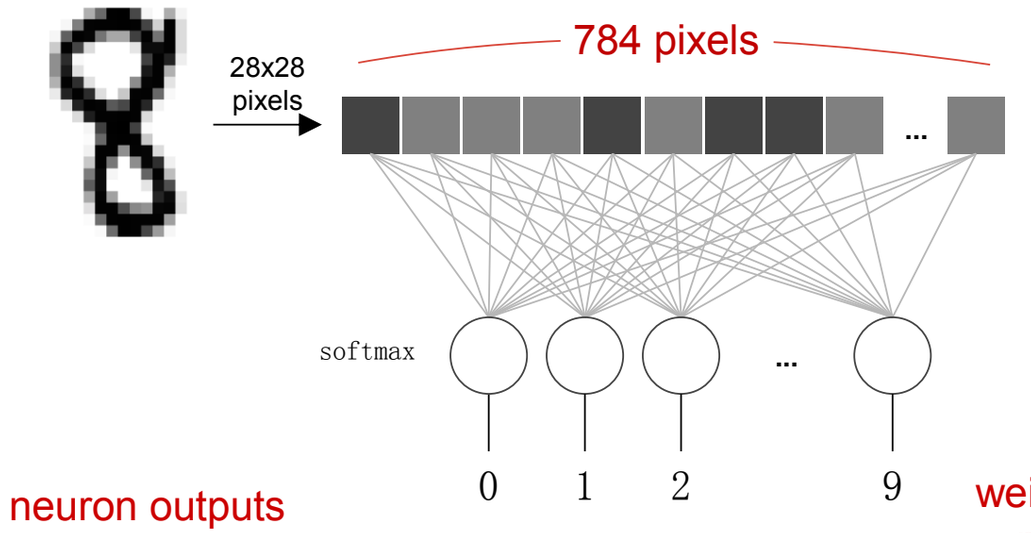


Output

- Classification: One-hot representation
- SoftMax
  - Boltzmann distribution: $e^{-E/kT}$
  - Takes vector of arbitrary values and maps to
    - vector with values in range 0 to 1
    - vector components sum to 1.
- Uses:
  - Prob output from Multiclass classification
  - Normalization of data.
  - …

The softmax layer

➢ **The output from the softmax layer is a set of probability distribution, positive numbers which sum up to 1.**



$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Cost Functions

Cost/Loss

- ***MSE***- Mean square error
    - Proven to give right y for given x.
- ***MAE***- Mean absolute error
    - Proven to give right median y for given x.
- For classification often train poorly:
    - Saturating outputs give small gradients.
    - Output for selected class is 1
    - Use ***cross-entropy*** for classification
- You can write your own log-likelihood cost function

The Cross-Entropy Cost Function

- **For classification problems, the Cross-Entropy cost function works better than quadratic cost function.**
- **We define the cross-entropy cost function for the neural network by:**

0	0	0	0	0	0	1	0	0	0

"one-hot" encoded ground truth

Cross entropy

$$C = -\sum y_i' \cdot \log\left(y_i'\right)$$

computed probabilities

this is a "6"

0.01	0.01	0.01	0.01	0.01	0.01	0.90	0.01	0.02	0.01
0	1	2	3	4	5	6	7	8	9

$$H(p,q) = -\sum_x p(x)\ \log q(x).$$

- In information theory, the cross entropy between two probability distributions p and q over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution q rather than the "true" distribution p

KL Divergence

$$D_{KL}(p||q) = \sum_{i=1}^{N} p(x_i) \cdot log \frac{p(x_i)}{q(x_i)}$$

- p is the truth
- q is your “coding” of truth
- D ~ number of extra bits needed to code p when starting code q
- $\Longrightarrow$ Minimize $D_{KL} \longrightarrow q$ encodes same information as p

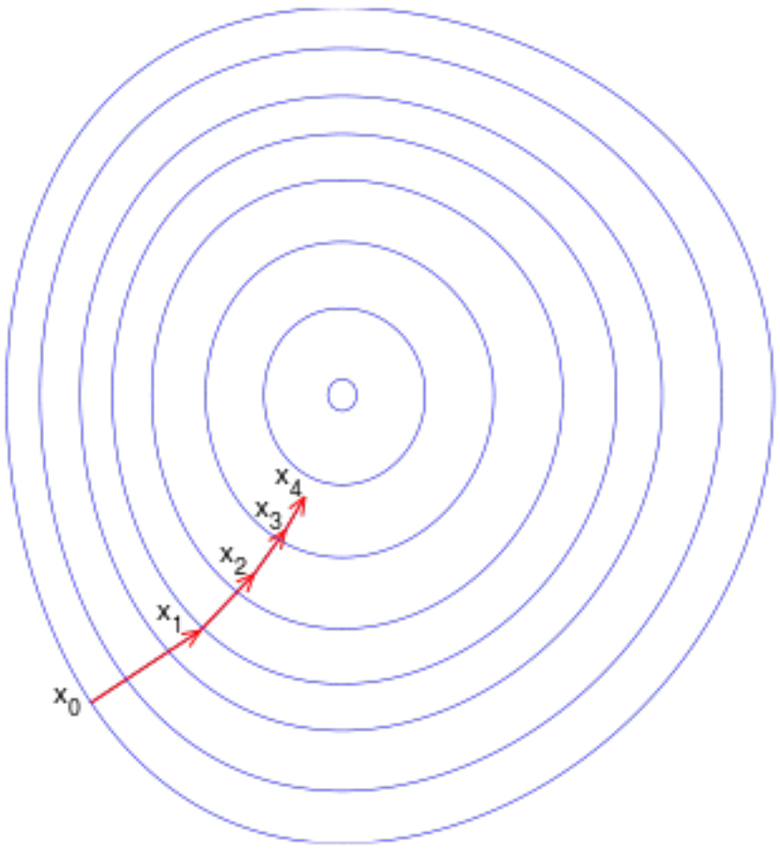
Optimizers

Training == Optimization

- Training = Minimizing cost function w.r.t. parameters **α**

$$C[F(\vec{X}_{train}|\vec{\alpha}), \vec{Y}_{train}] \equiv C(\vec{\alpha})$$

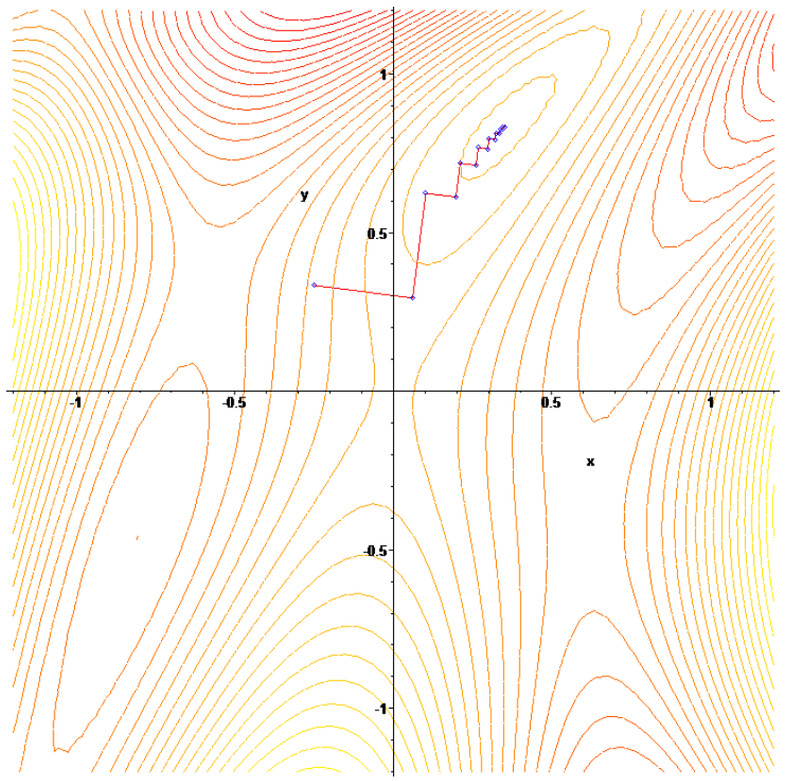
- Gradient Decent (Newton's Method):



- Gradient points to direction of maximal change.

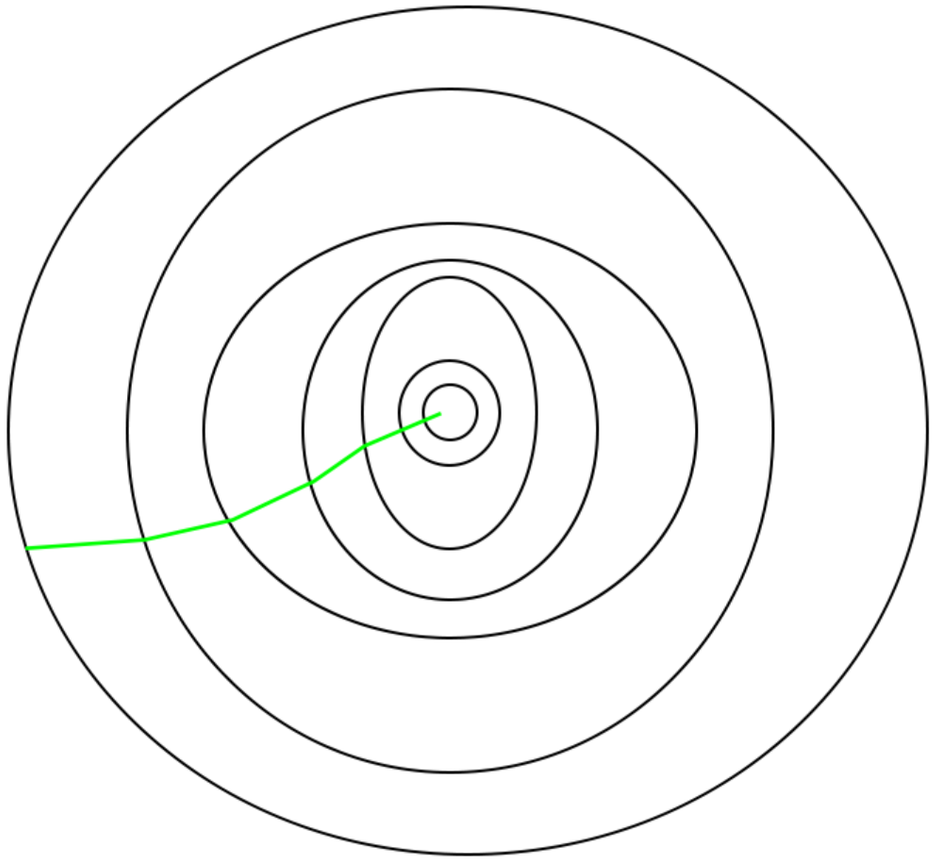
- Iterate (ε sets the step size == *Learning Rate*)

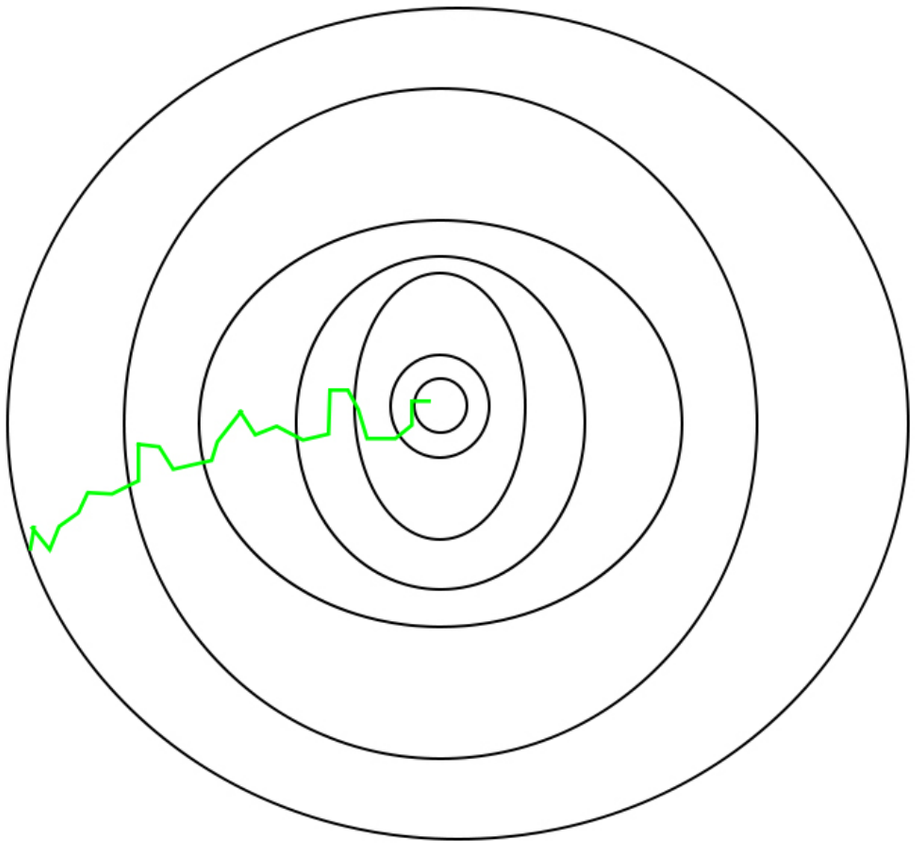
$$\vec{\alpha}_{i+1} = \vec{\alpha}_i - \epsilon \nabla C(\vec{\alpha})$$



Stochastic Gradient Decent

- Inefficient to compute the gradient on full dataset ⟶ take few steps.
- Approximate the gradient on a small subset (a batch) ⟶ take more steps
    - Noisy gradients… but faster (computationally) convergence.
- Number of Examples = Batch Size * Number of Batches
- Epoch = 1 pass through all examples.

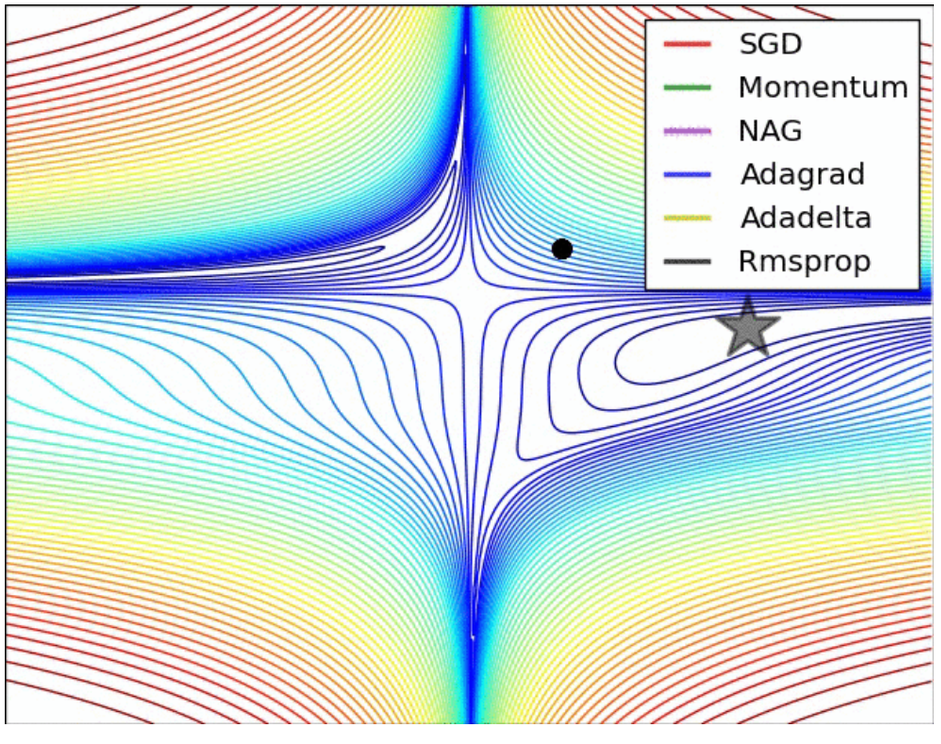




Learning Rate, Decay, Momentum

$$\vec{\alpha}_{i+1} = \vec{\alpha}_i - \epsilon \nabla C(\vec{\alpha})$$

- ε is the learning rate ~ the step size… a hyper parameter.
- May be beneficial to take big steps in the beginning, small steps near the minimum.
  - Decay: at epoch *i*, $\varepsilon_i \longrightarrow \varepsilon_{i-1} / (1 + \delta)$ a hyperparameter
- Gradient can be noisy
  - Momentum: add a bit of the previous batch gradient to gradient of current batch.
- Lots of difference strategies… again, a hyperparameter.

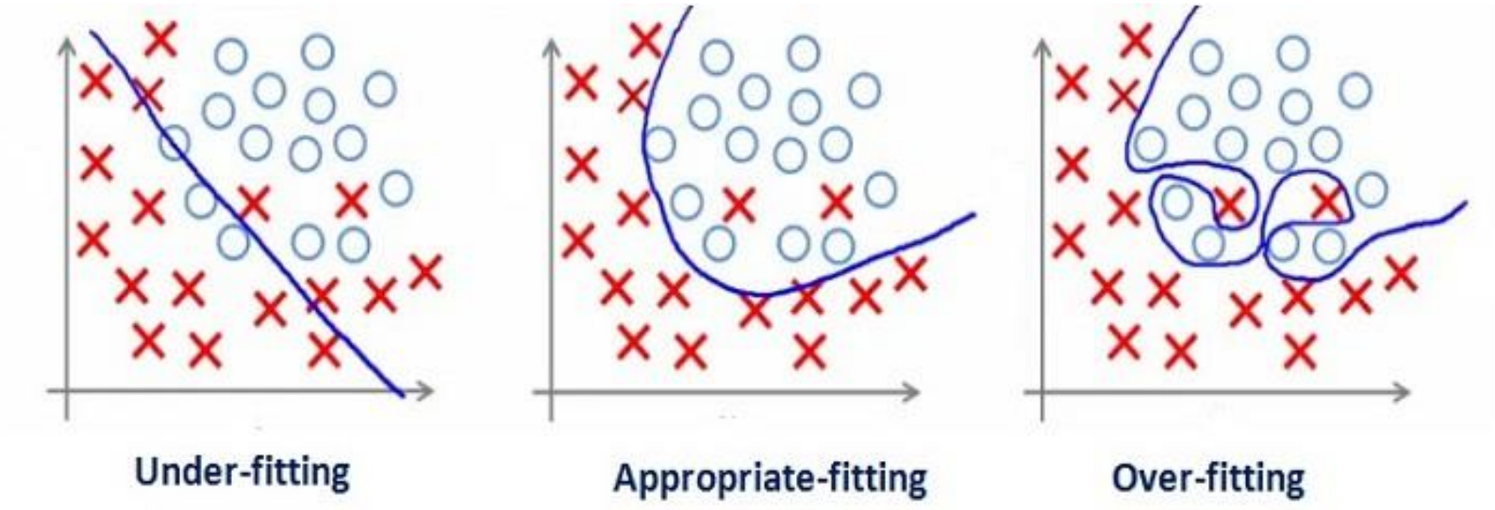


Training

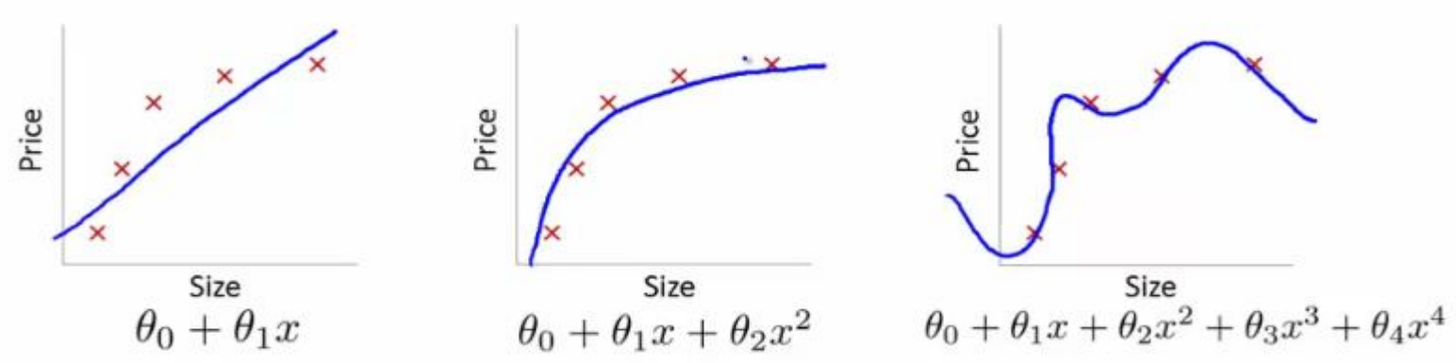
Over Fitting

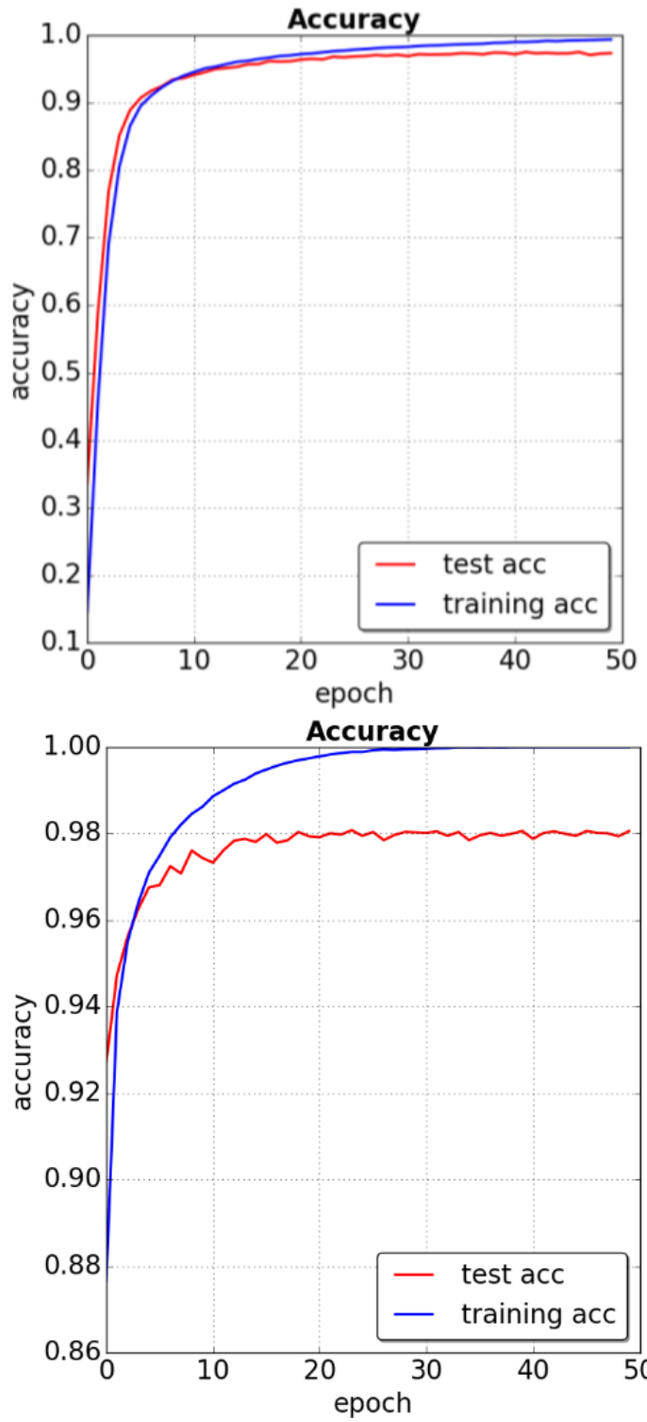
- **Overfitting occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. A model that has been overfit has poor predictive performance, as it overreacts to minor fluctuations in the training data.**

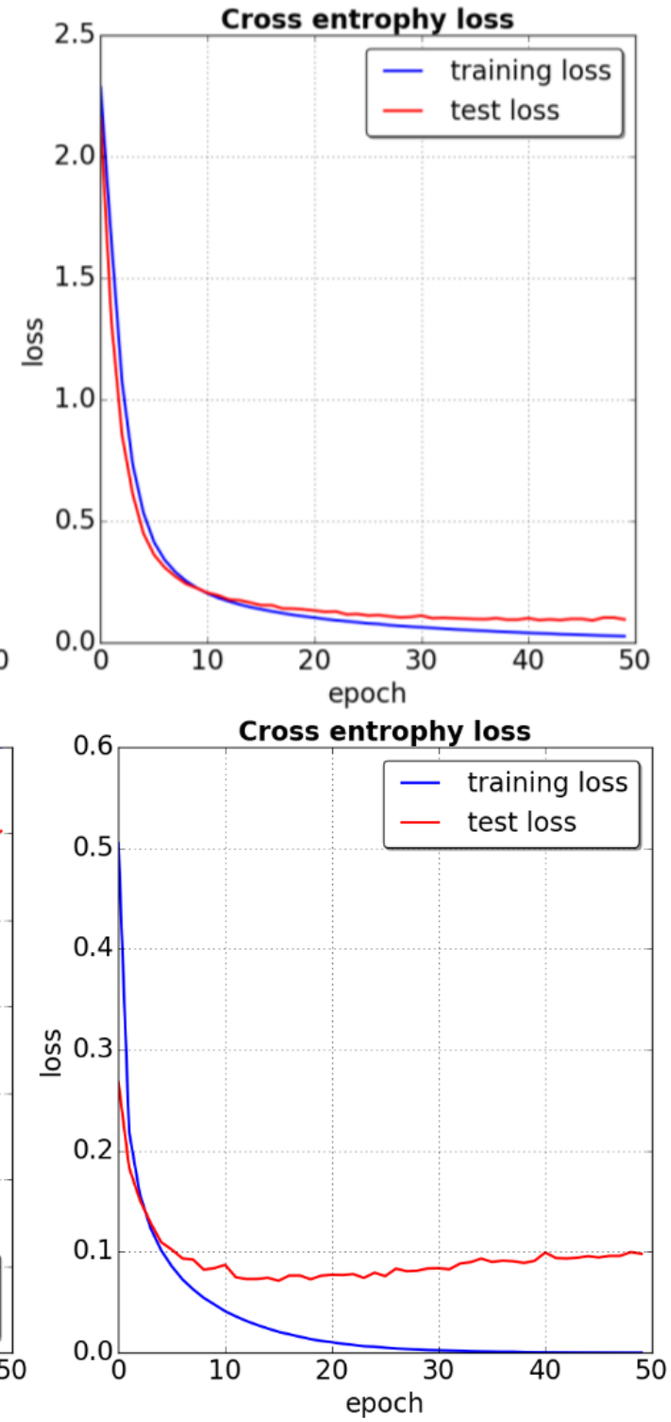
Classification:



Regression:




Accuracy
accuracy
epoch
test acc
training acc
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50

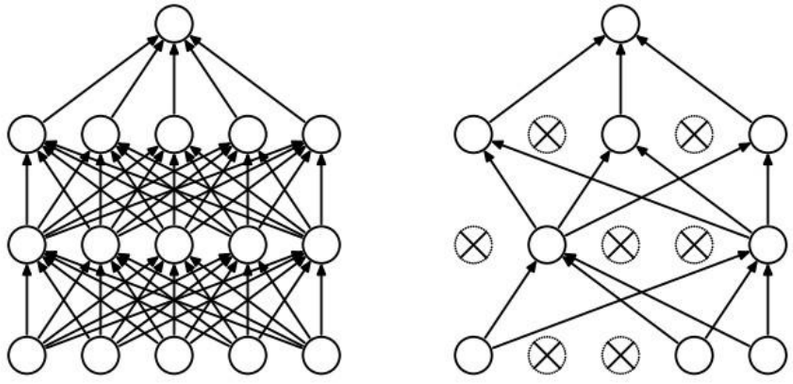

Cross entrophy loss
loss
epoch
training loss
test loss
2.5
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50

Accuracy
accuracy
epoch
test acc
training acc
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
10
20
30
40
50

Cross entrophy loss
loss
epoch
training loss
test loss
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
10
20
30
40
50

Regularization - Dropout

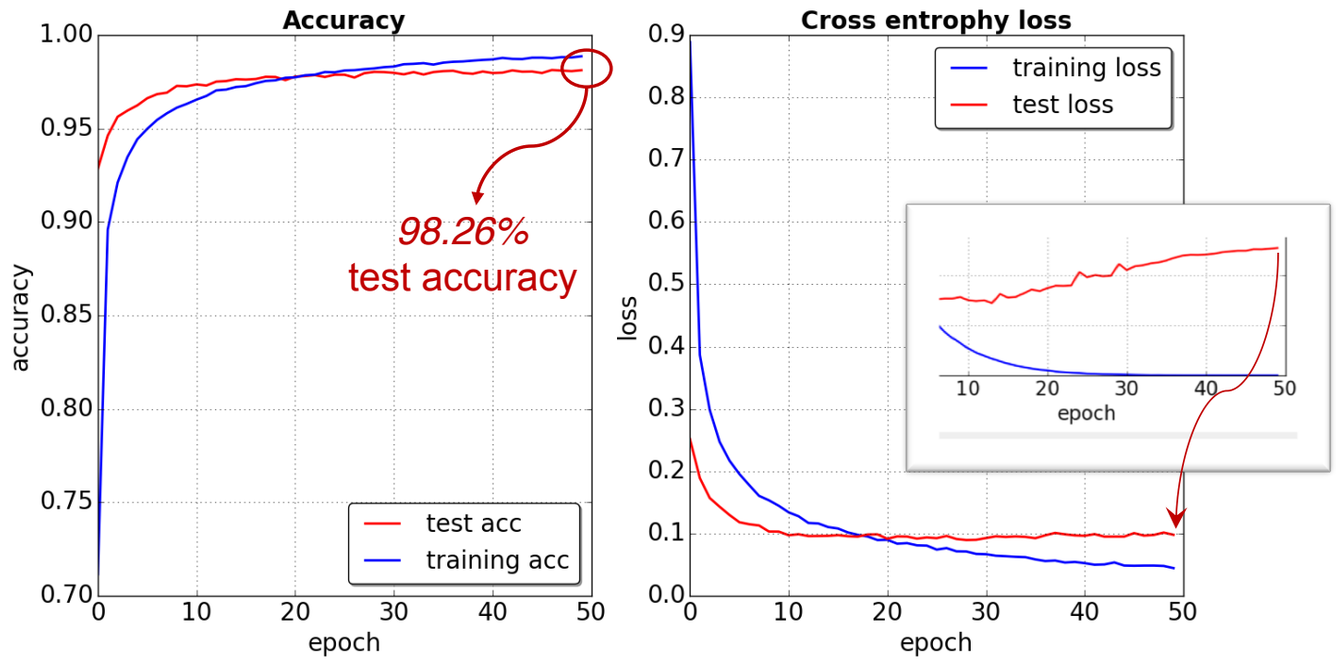
- **Dropout is an extremely effective, simple and recently introduced regularization technique by Srivastava et al (2014).**



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

- **While training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise.**
- **It is quite simple to apply dropout in Keras.**

```
# apply a dropout rate 0.25 (drop 25% of the neurons)
model.add(Dropout(0.25))
```

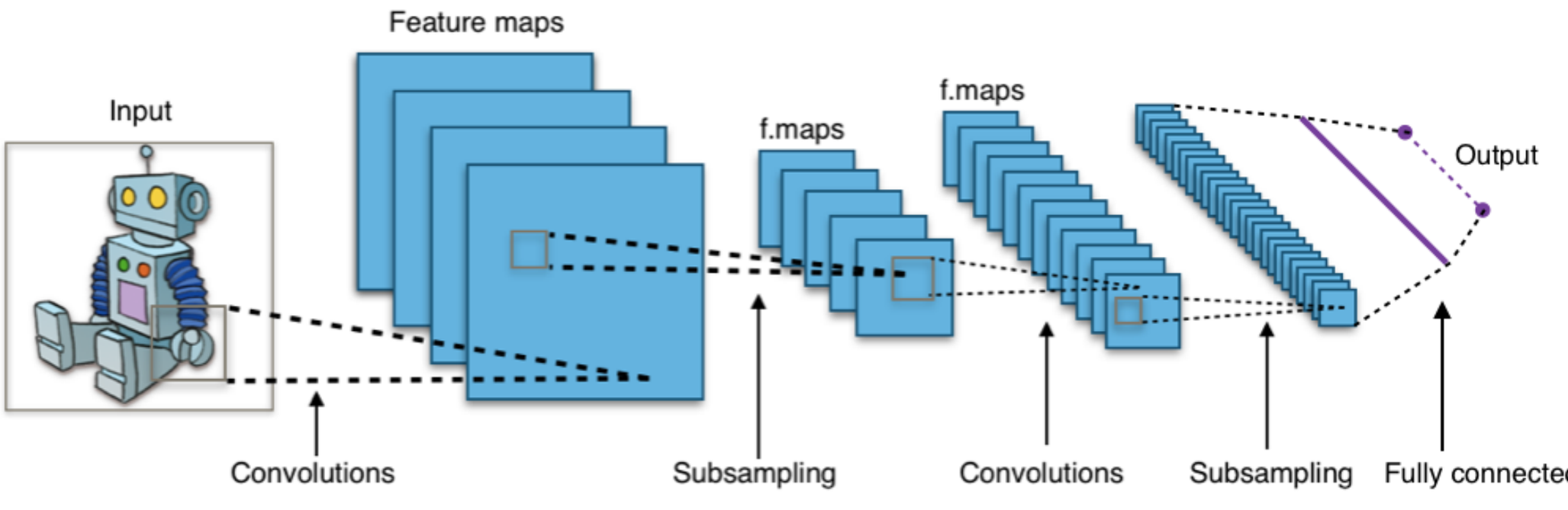


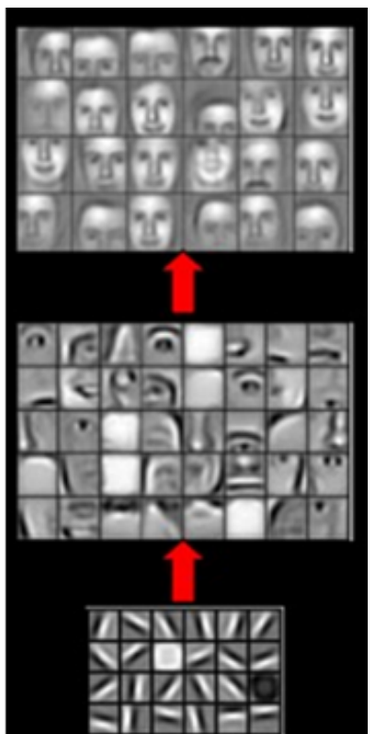
Architectures

CNNs

Convolutional NNs

- Inspired by visual cortex
  - ***Input***: Raw data… for example 1D = Audio, 2D = Images, 3D = Video
  - ***Convolutions*** ~ learned feature detectors
  - ***Feature Maps -*** One per convolution kernel
  - ***Pooling*** - dimension reduction / invariance
  - ***Stack***: Deeper layers recognize higher level concepts.
- Hyperparameters:
  - Number, Size, and stride of convolutions
  - Pooling sub-sampling and method (average, max, …)
  - Number of times…

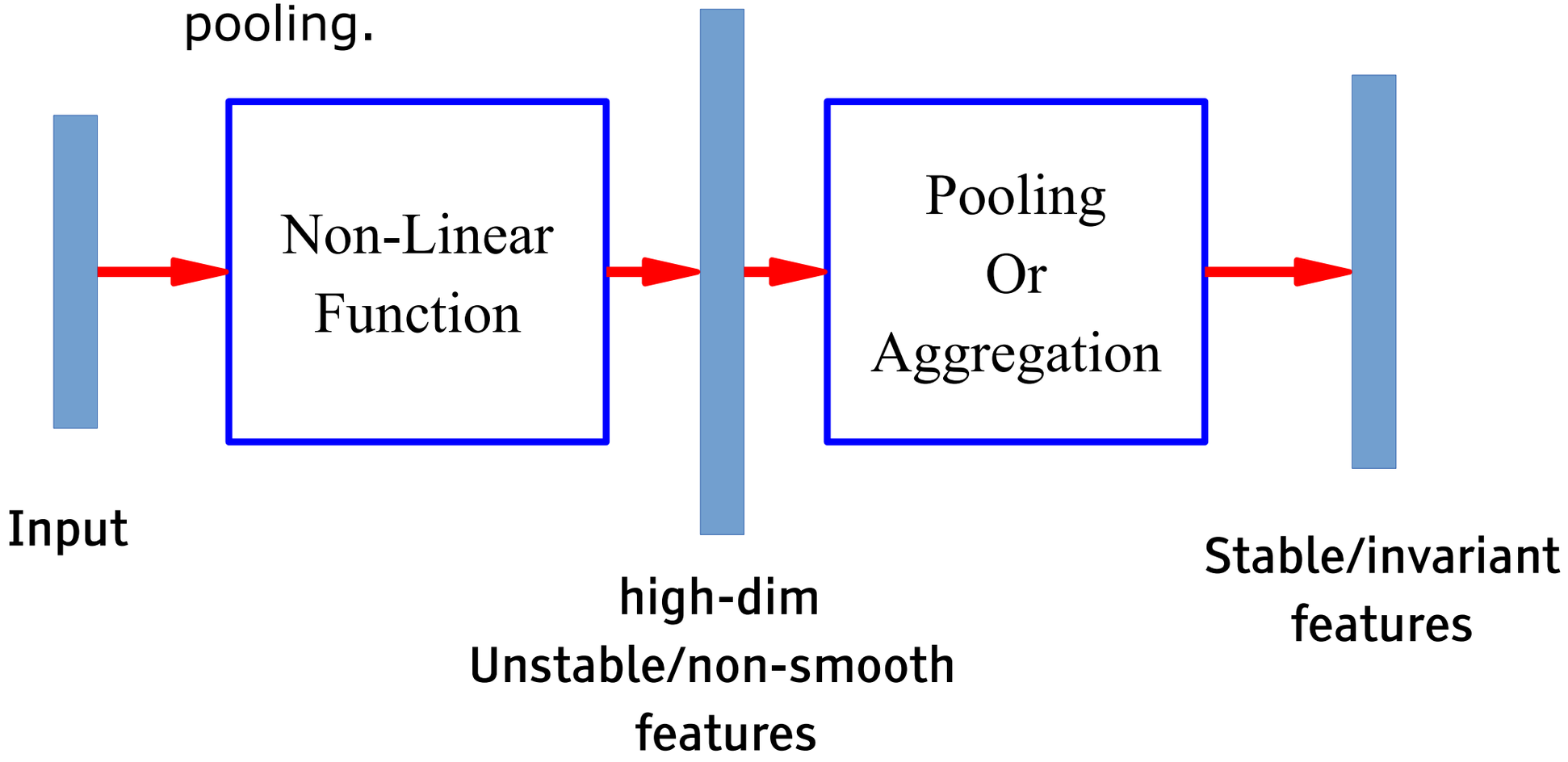




Basic Idea for Invariant Feature Learning

Y LeCun
MA Ranzato

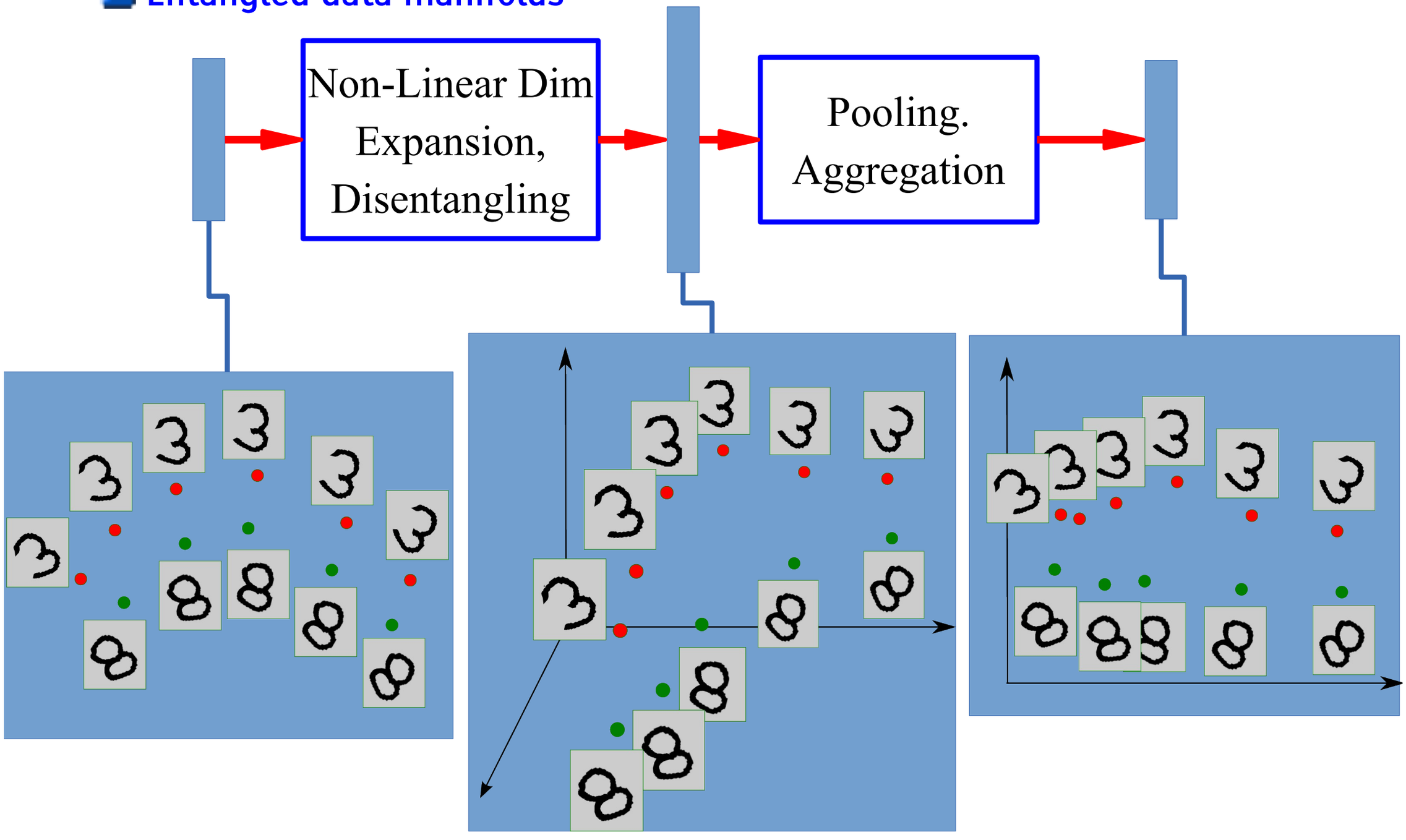
- Embed the input non-linearly into a high(er) dimensional space
  - In the new space, things that were non separable may become separable
- Pool regions of the new space together
  - Bringing together things that are semantically similar. Like pooling.



Non-Linear Expansion → Pooling

Y LeCun
MA Ranzato

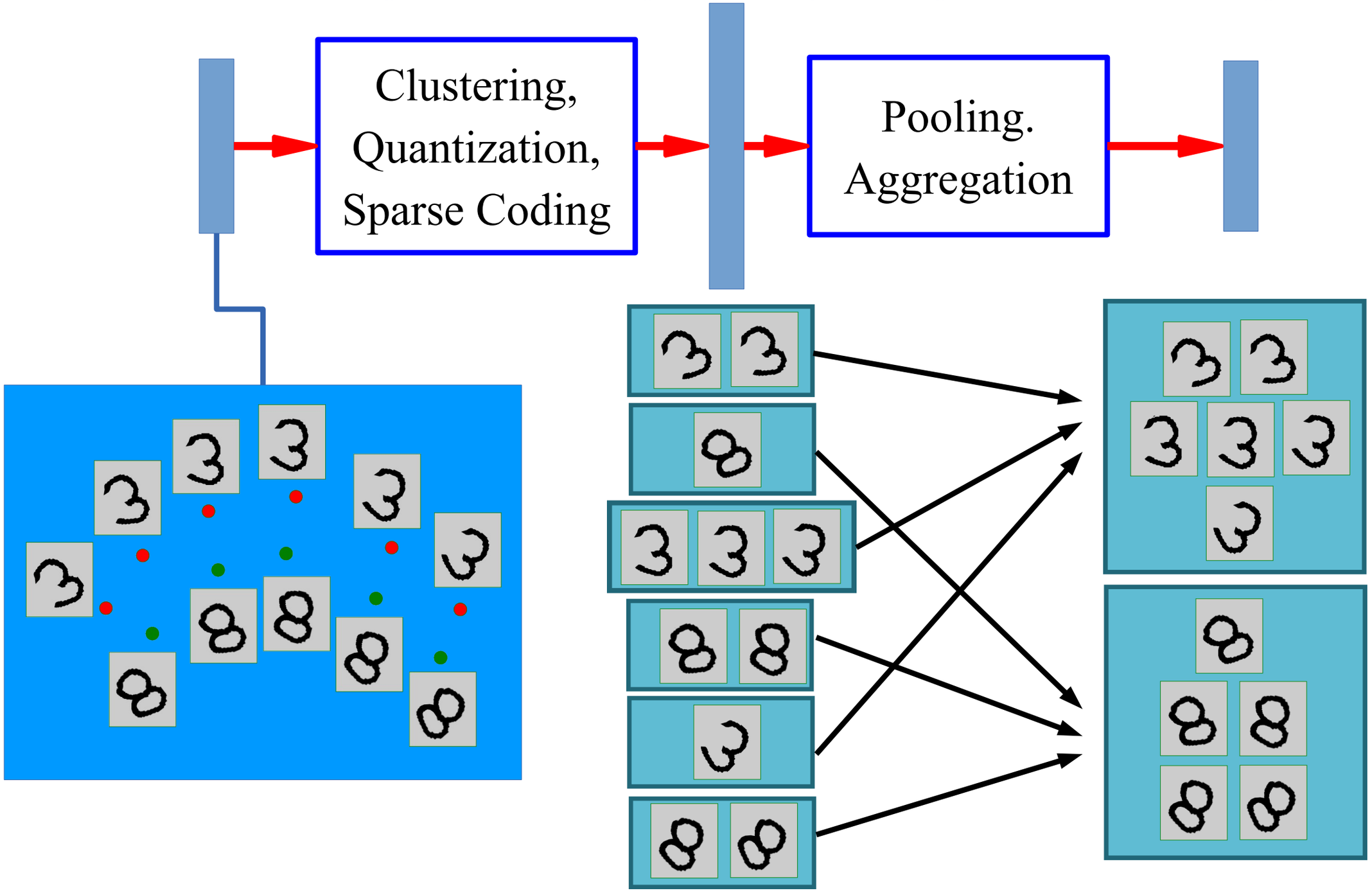
- Entangled data manifolds



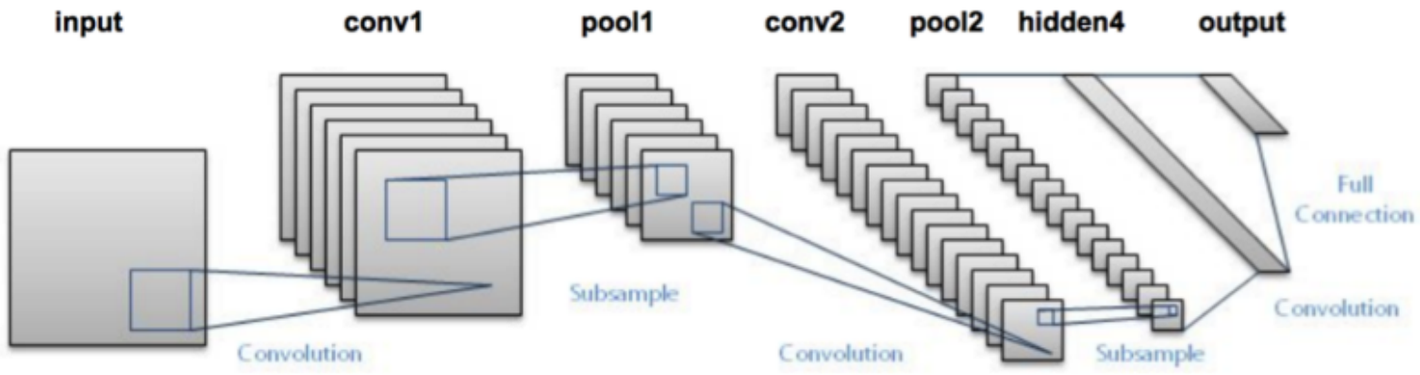
Sparse Non-Linear Expansion → Pooling

Y LeCun
MA Ranzato

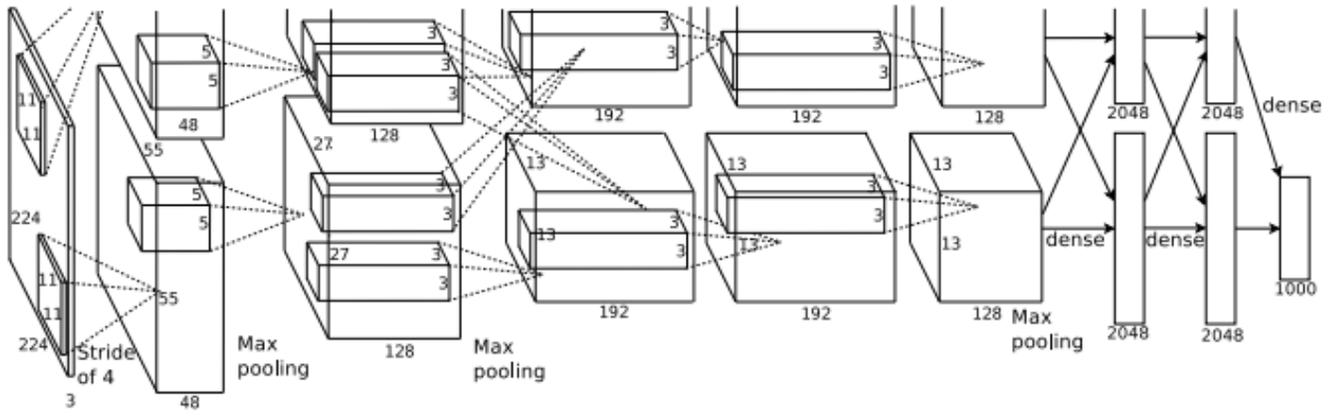
- Use clustering to break things apart, pool together similar things



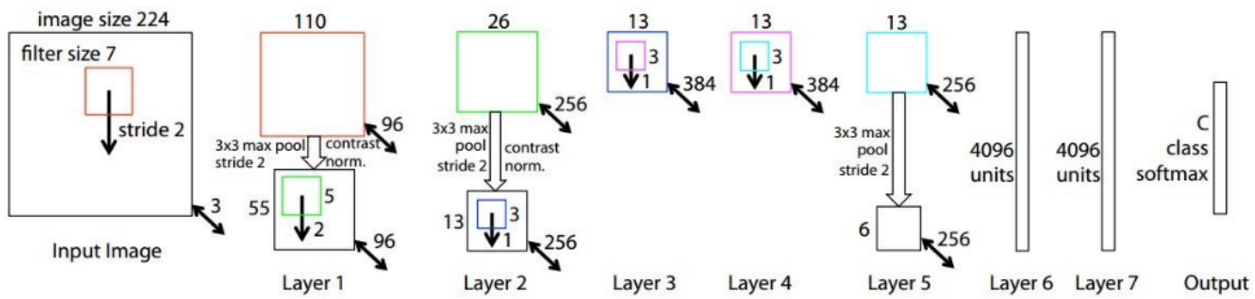
LeNet-5 (1998)



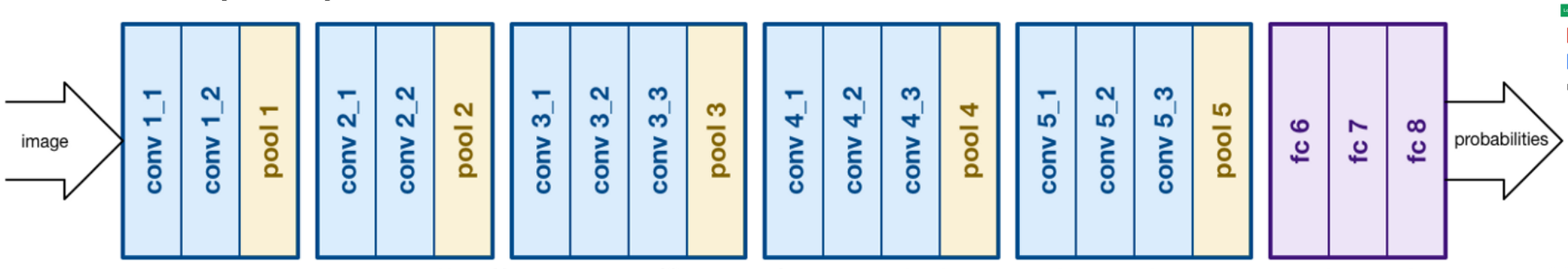
AlexNet (2012)



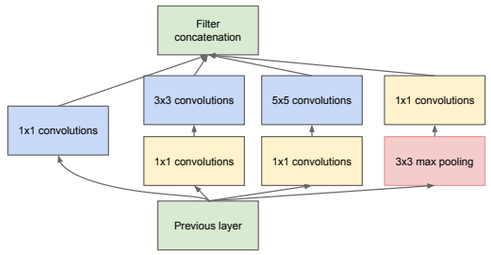
ZFNet(2013)

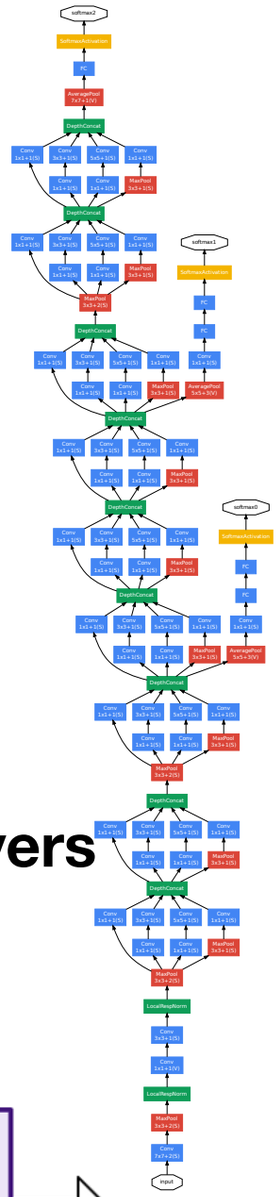


VGGNet (2014)



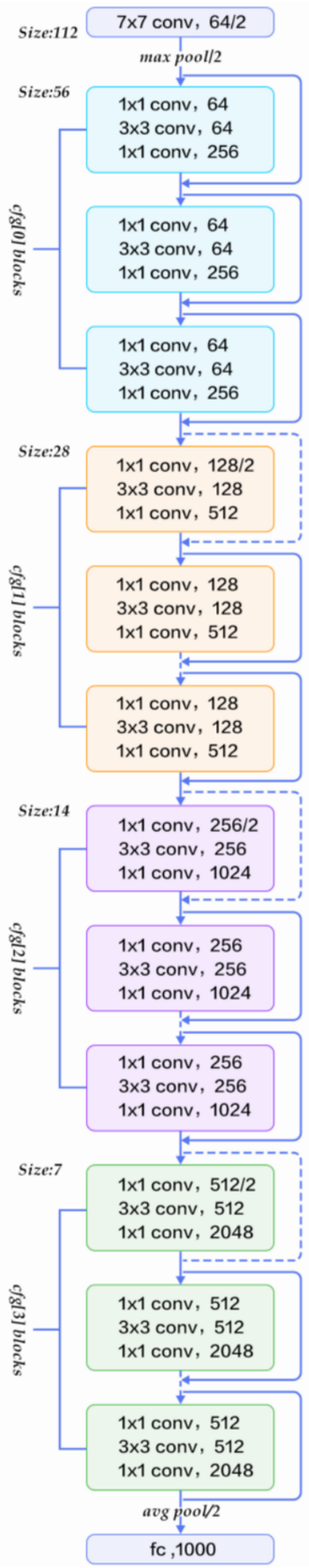
GoogleNet/Inception(2014)





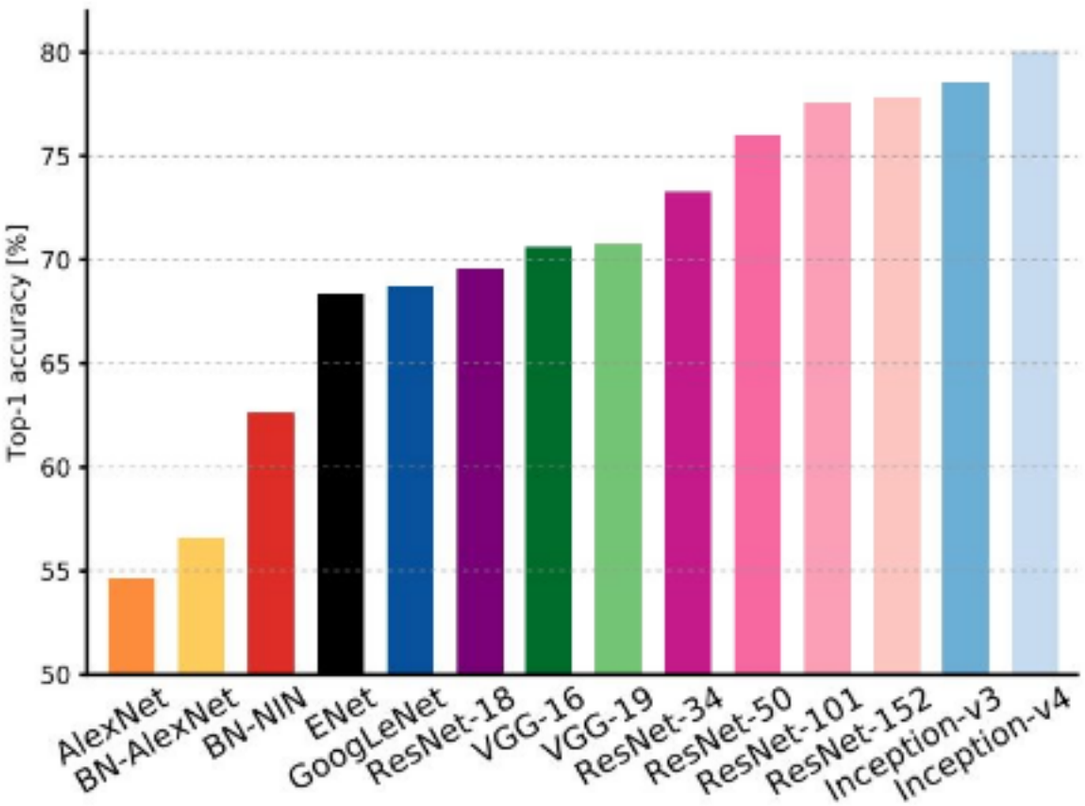
22 Layers

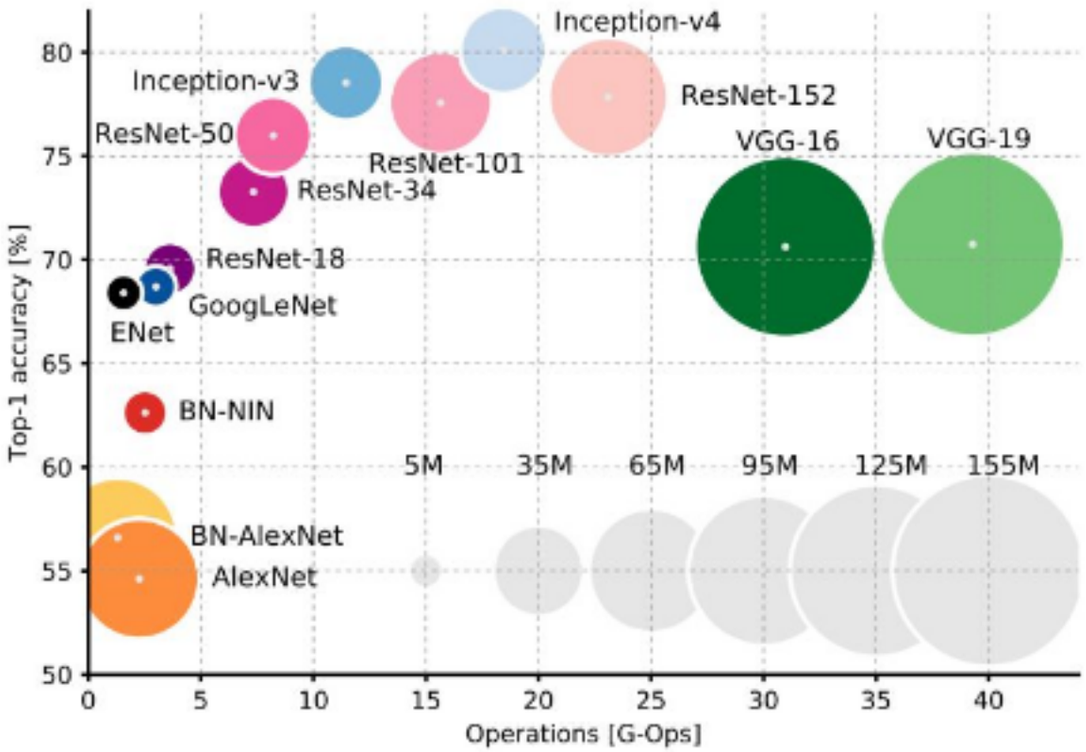
ResNet(2015)



152 Layers

https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

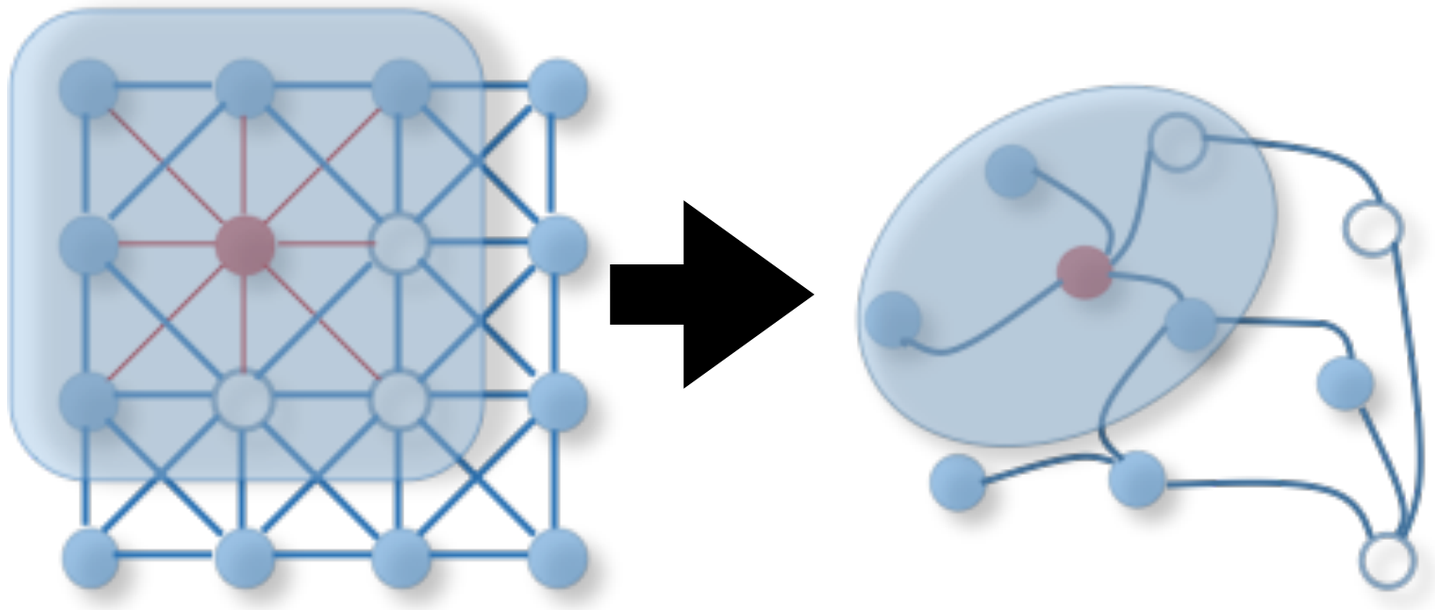




An Analysis of Deep Neural Network Models for Practical Applications, 2017.

https://medium.com/@sidereal/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

Graph Convolutional NN



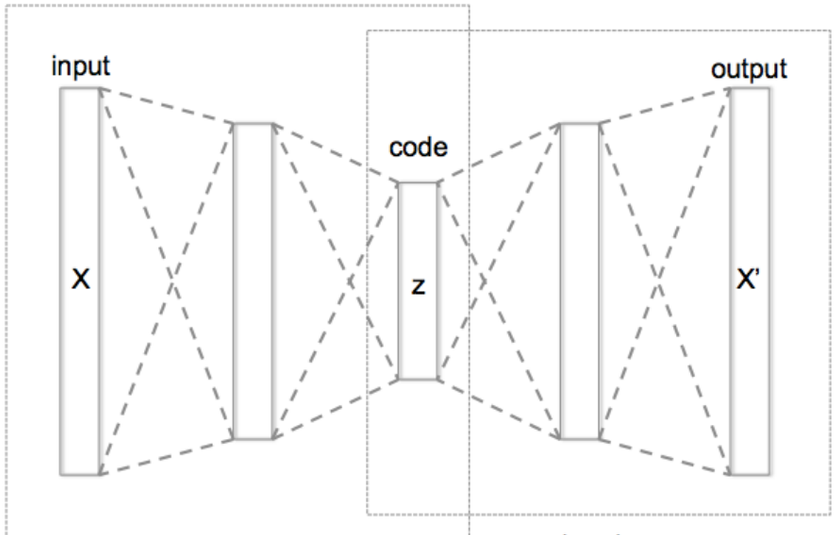
https://arxiv.org/pdf/1901.00596.pdf

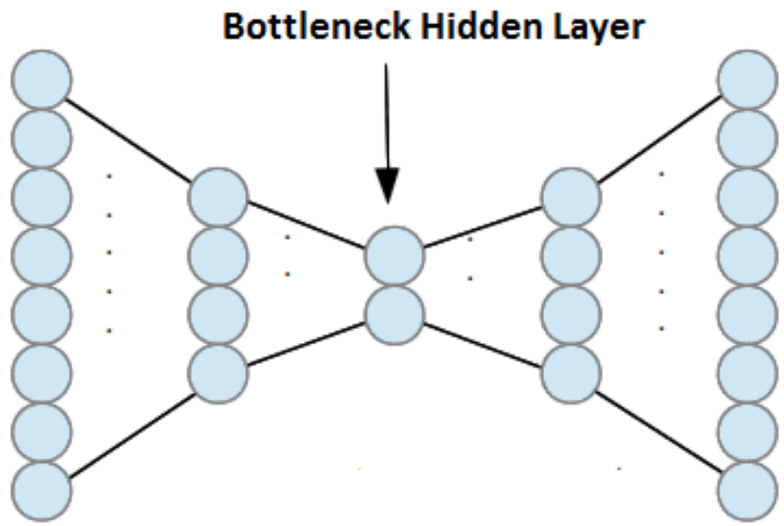
Transfer Learning

- Training Challenges
    - Can take a long time… maybe weeks.
        - Parallelization can help… if you have lots of resources.
    - You may not have enough training data for task you want to accomplish.
- Transfer learning: I want to a DNN to perform task A on sample 1.
    - Train on large dataset B of same type as A (e.g. images) on a specific task 2…
        - or use pre-trained model.
    - Modify output task B $\longrightarrow$ A.
    - Fine-tune: train to perform task A on sample 1.
    - $\Longrightarrow$ Train faster, get better results, use smaller training data.
- You can find pre-trained models for image recognition, NLP, and many other tasks.

Semi-supervised Learning

- Basic idea: Train network to ***reproduce the input***.
- Example: ***Auto-encoders***
  - ***De-noising auto-encoders***: add noise to input only.
  - ***Sparse auto-encoders***:
    - ***Sparse latent (code) representation*** can be exploited for ***Compression, Clustering, Similarity testing,*** …
    - ***Anomaly Detection***
      - Reconstruction Error
      - Outliers in latent space
    - ***Transfer Learning***
      - Small labeled training sample?
        - Train auto-encoder on large unlabeled dataset (e.g. data).
        - Train in latent space on small labeled data. (e.g. rare signal MC).
- Easily think of a dozen applications.

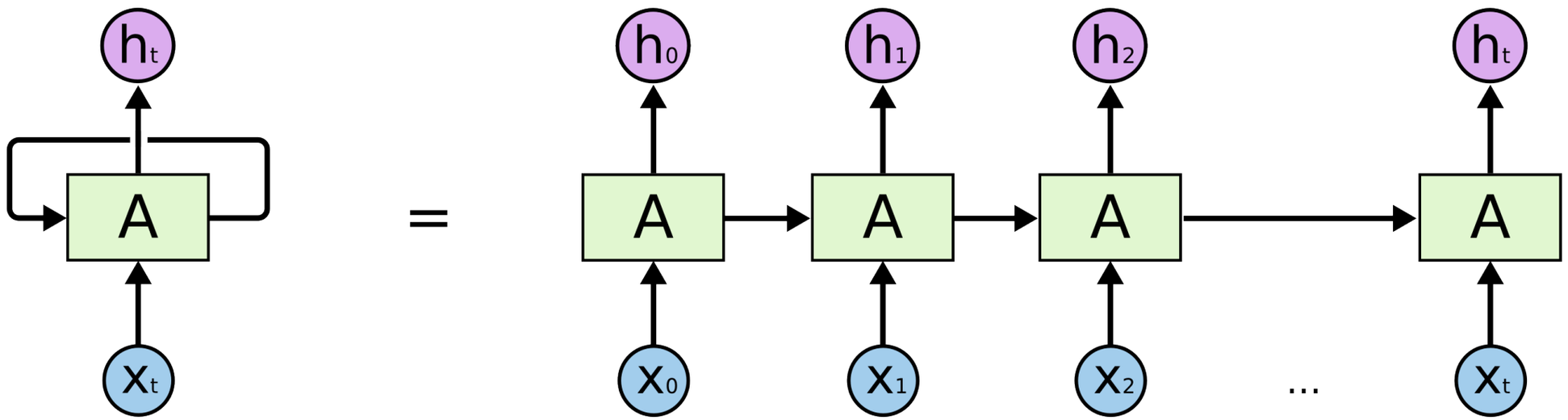




Recurrent DNNs

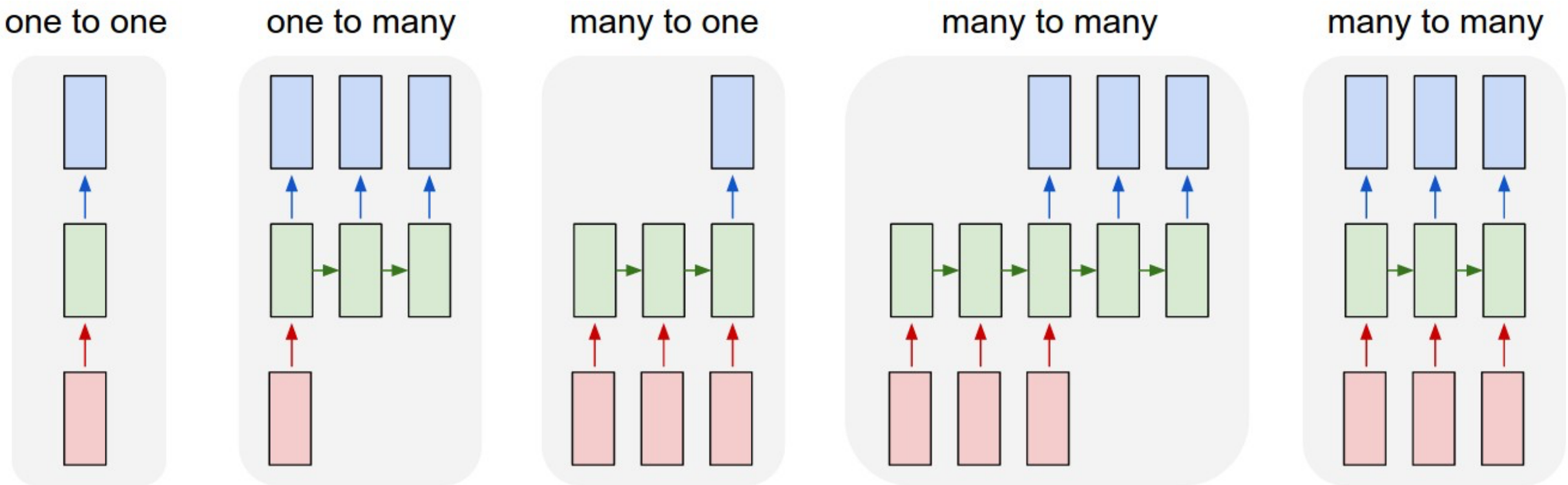
Motivation

- DNN inputs:
  - Fixed size: images, video, raw data from detector…
  - Variable size: audio, text, particles in event… sequences.
- Usual NNs map input to output.
  - No memory of previous information.
- Recurrent NN: feed some output back into self.
- DNNs can represent arbitrary functions… RNNs can represent arbitrary programs.



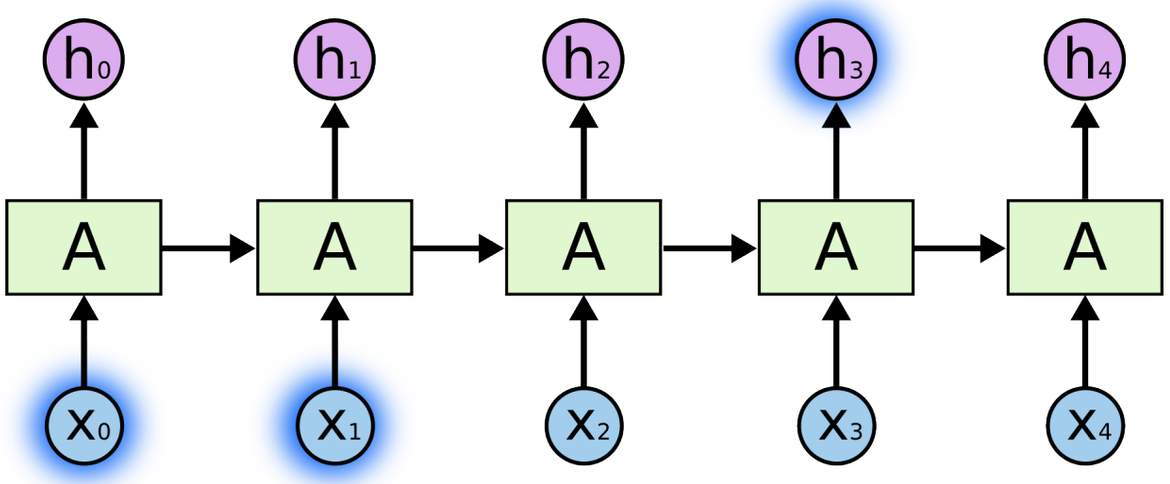
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

RNN input/output



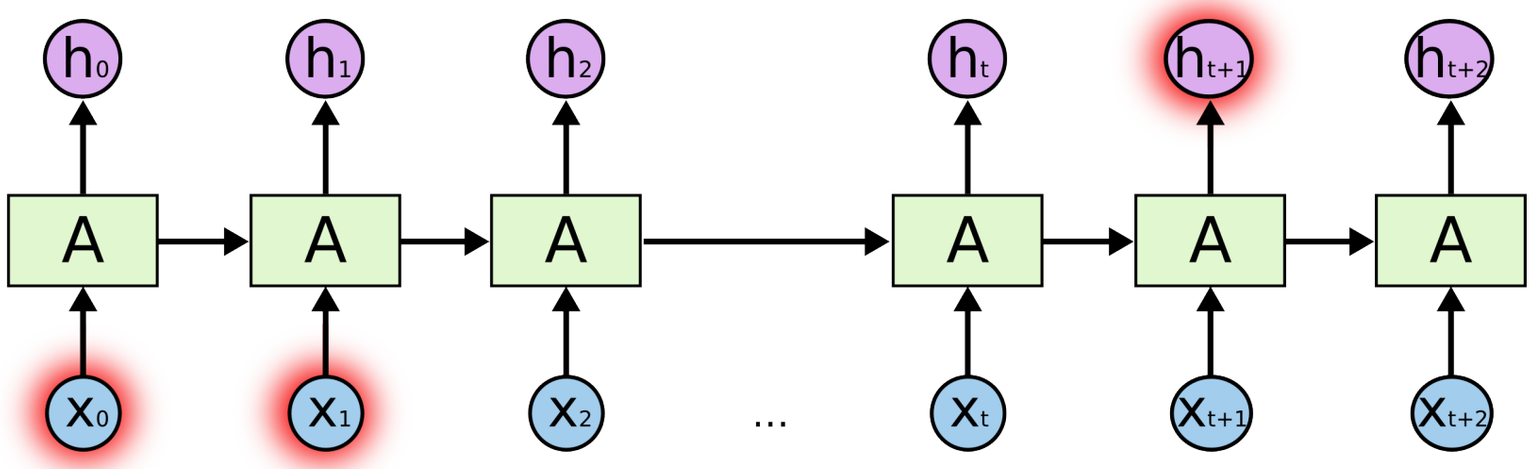
http://karpathy.github.io/2015/05/21/rnn-effectiveness/

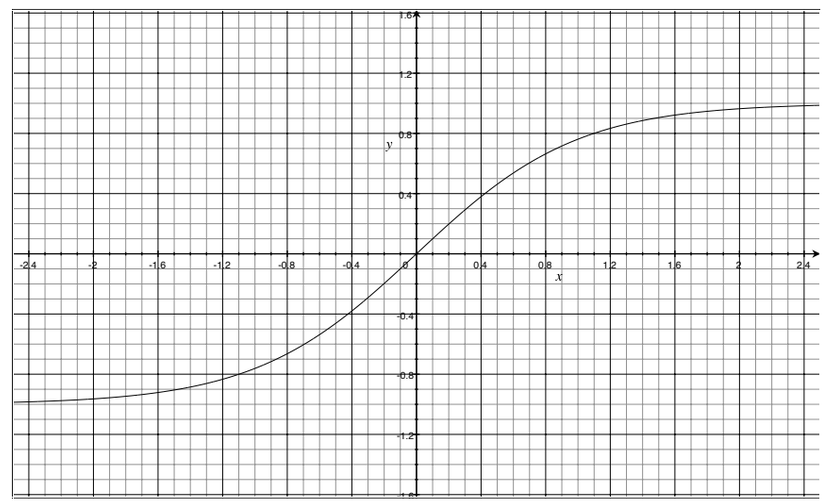
Basic RNN

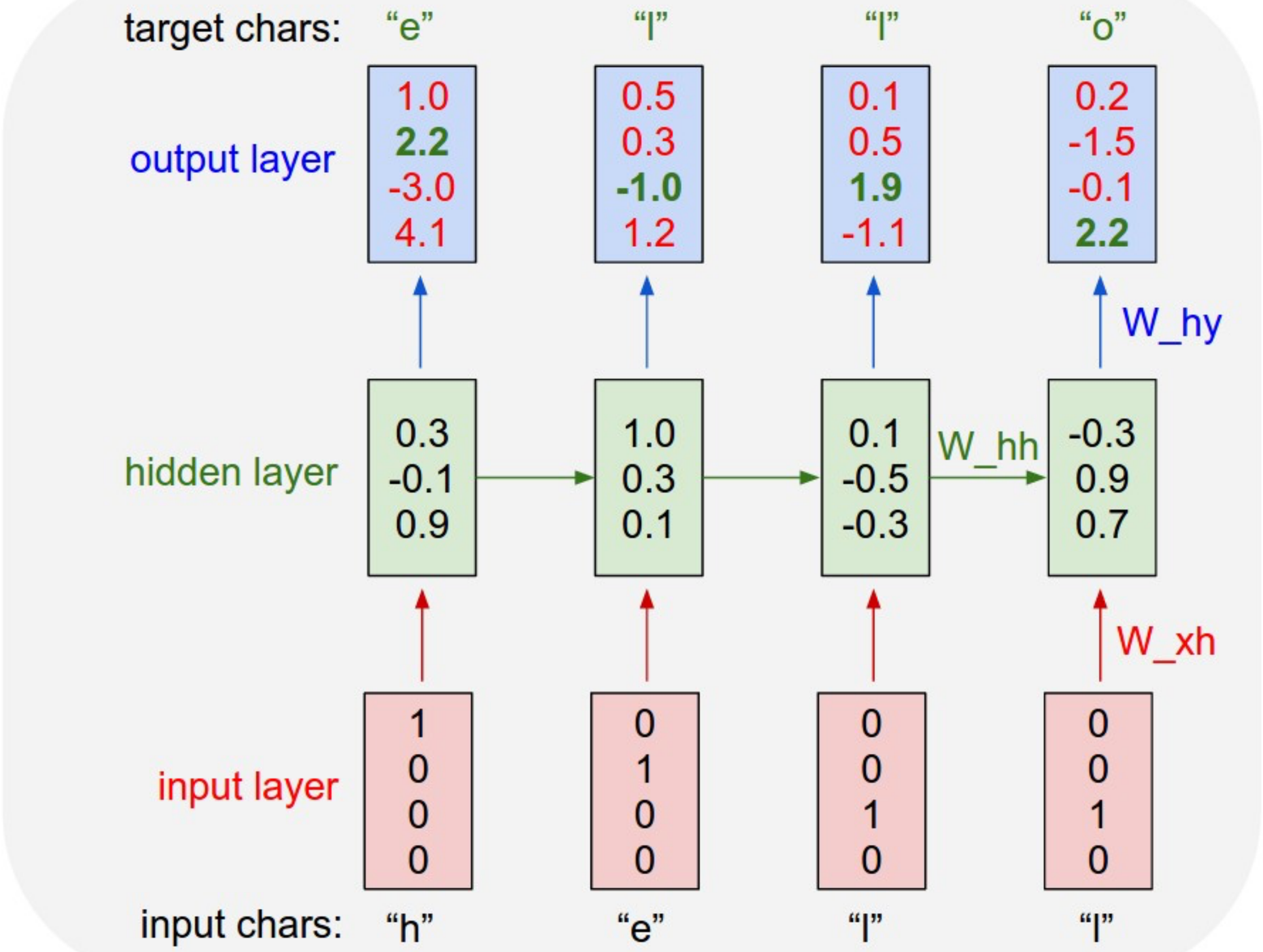


$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y = W_{hy} \cdot h_t$$

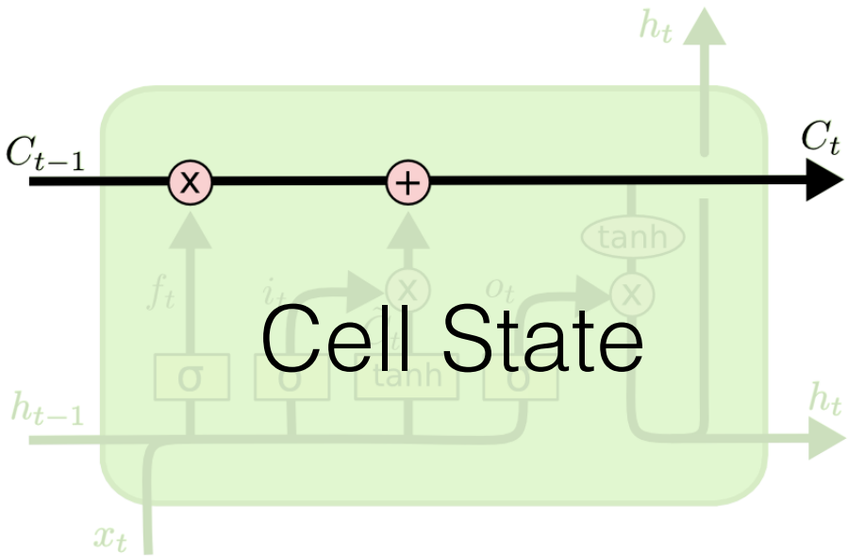




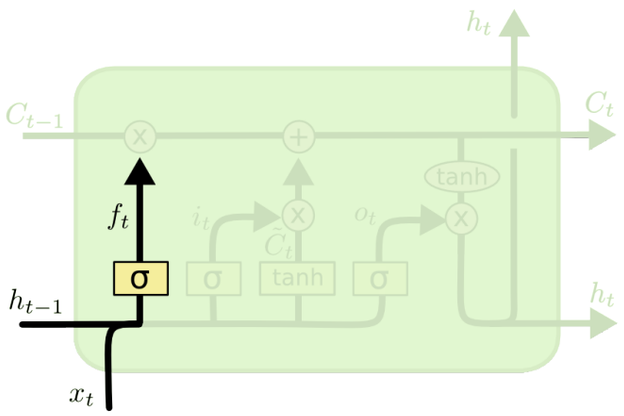


Sequence Predictor

LSTM

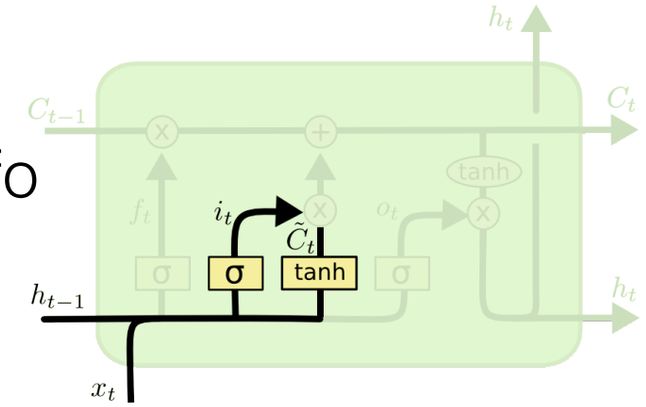


Forget



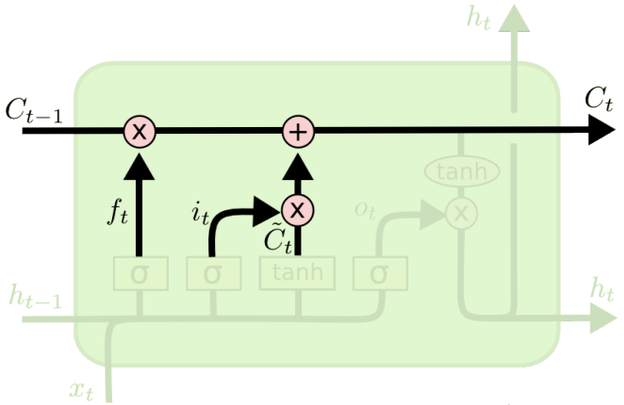
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

New info



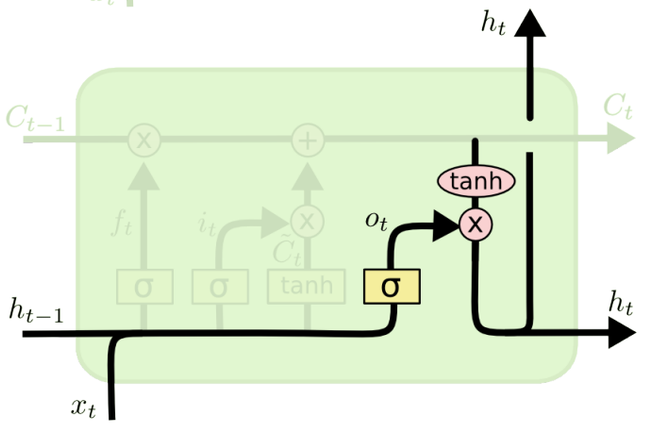
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

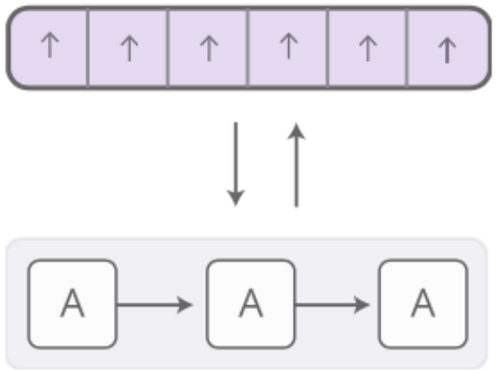
Construct output



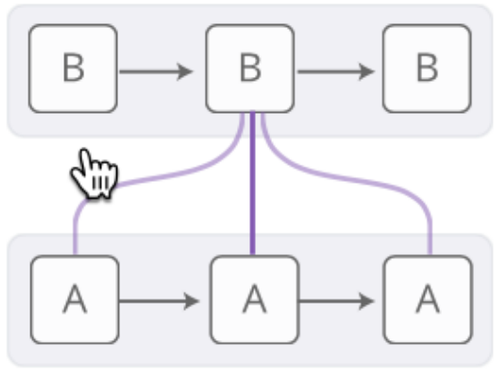
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

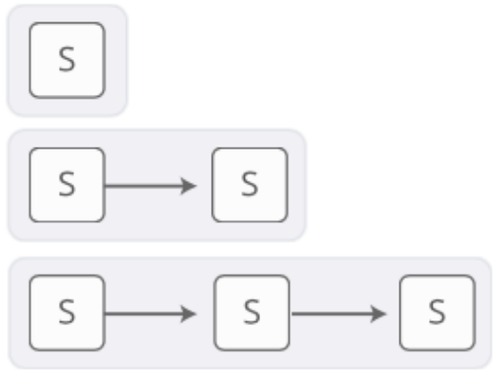
Other RNNs



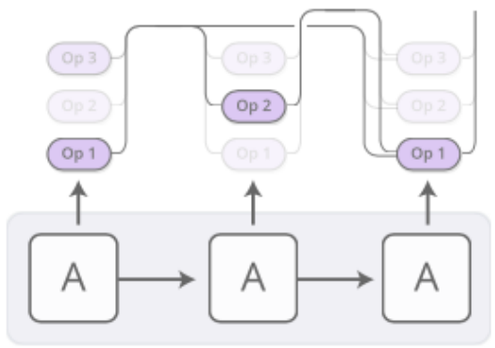
Neural Turing Machines
have external memory that they can read and write to.



Attentional Interfaces
allow RNNs to focus on parts of their input.



Adaptive Computation Time
allows for varying amounts of computation per step.

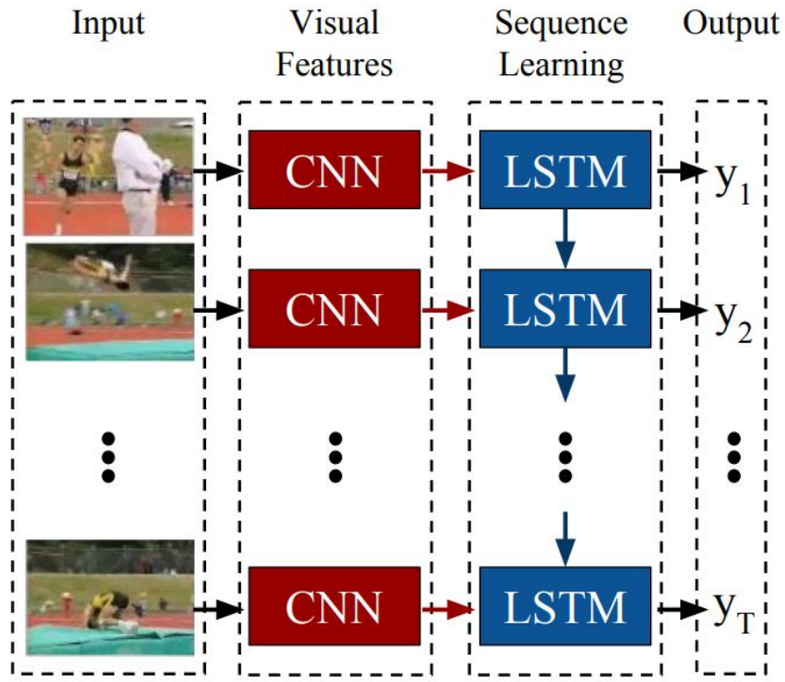


Neural Programmers
can call functions, building programs as they run.

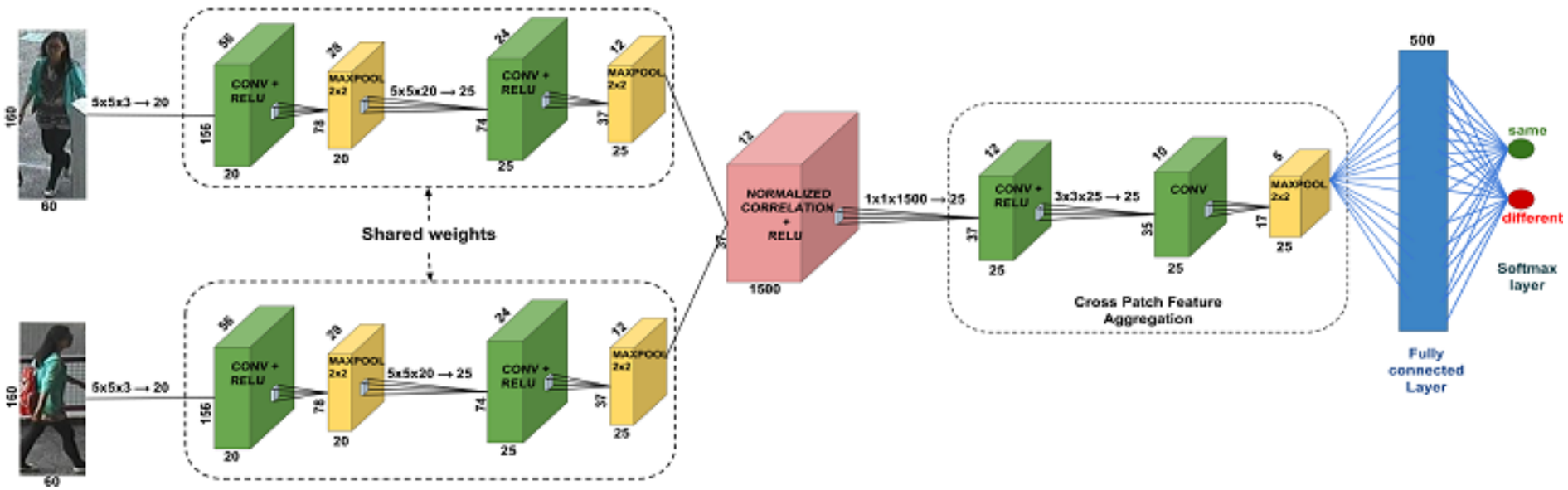
Combining Networks

Combining Architectures

- Different data are naturally suited for different architectures:
  - Audio: 1-D CNN, RNN
  - Image: 2D CNN
  - Text: RNN
- You can combine architectures:
  - Video: 2D CNN on frames → RNN in time.
- You can simultaneously look at multiple types of data:
  - Siamese: 2 parallel data inputs (same or different types)

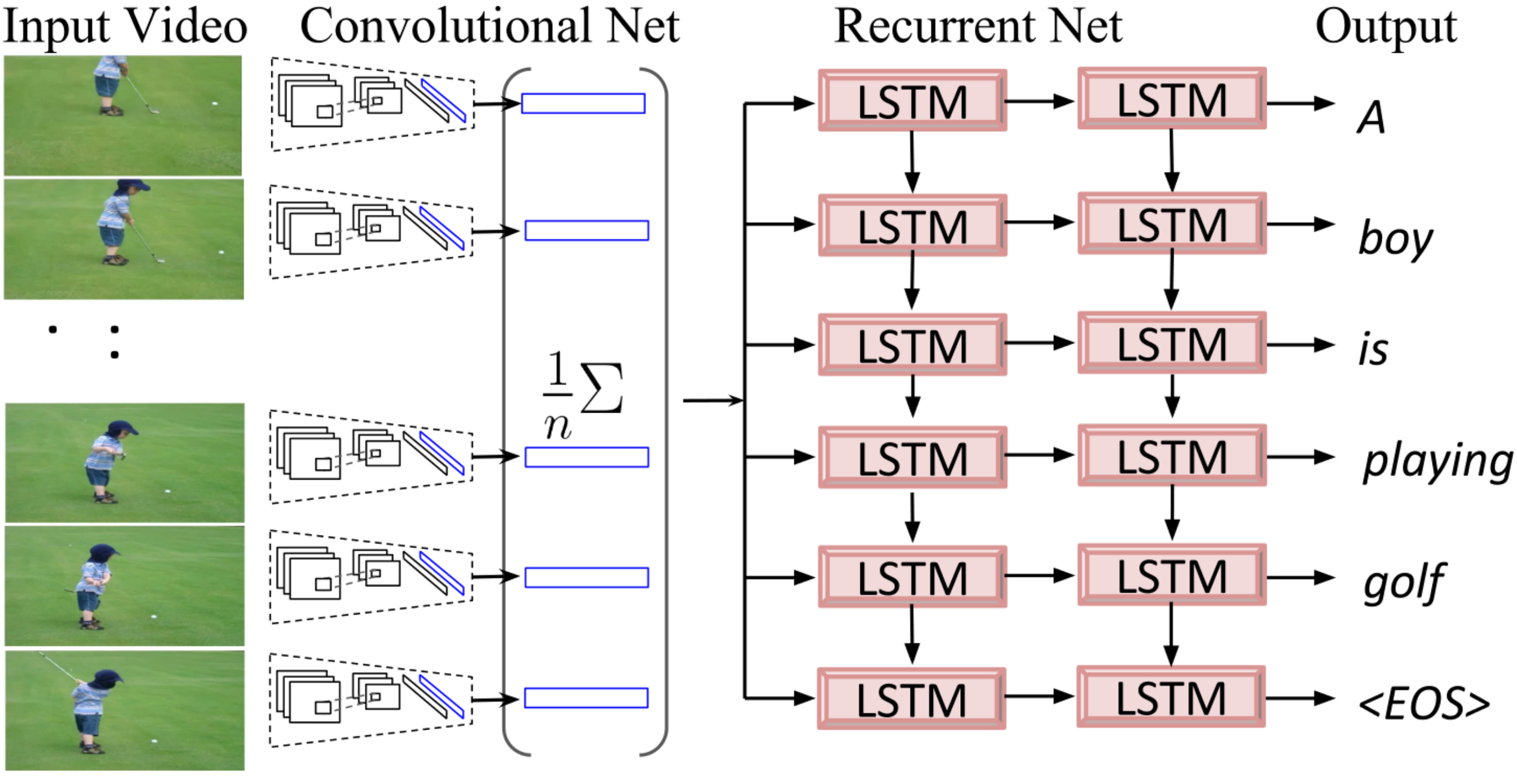


http://cs231n.stanford.edu/slides/2018/cs231n_2018_ds08.pdf



https://www.codeproject.com/Articles/1253224/Keras-Implementation-of-Siamese-like-Networks

Example: Captioning



https://www.cs.utexas.edu/~vsub/naacl15_project.html

Generative Models

VAE: Variational Autoencoders

Auto-Encoding Variational Bayes

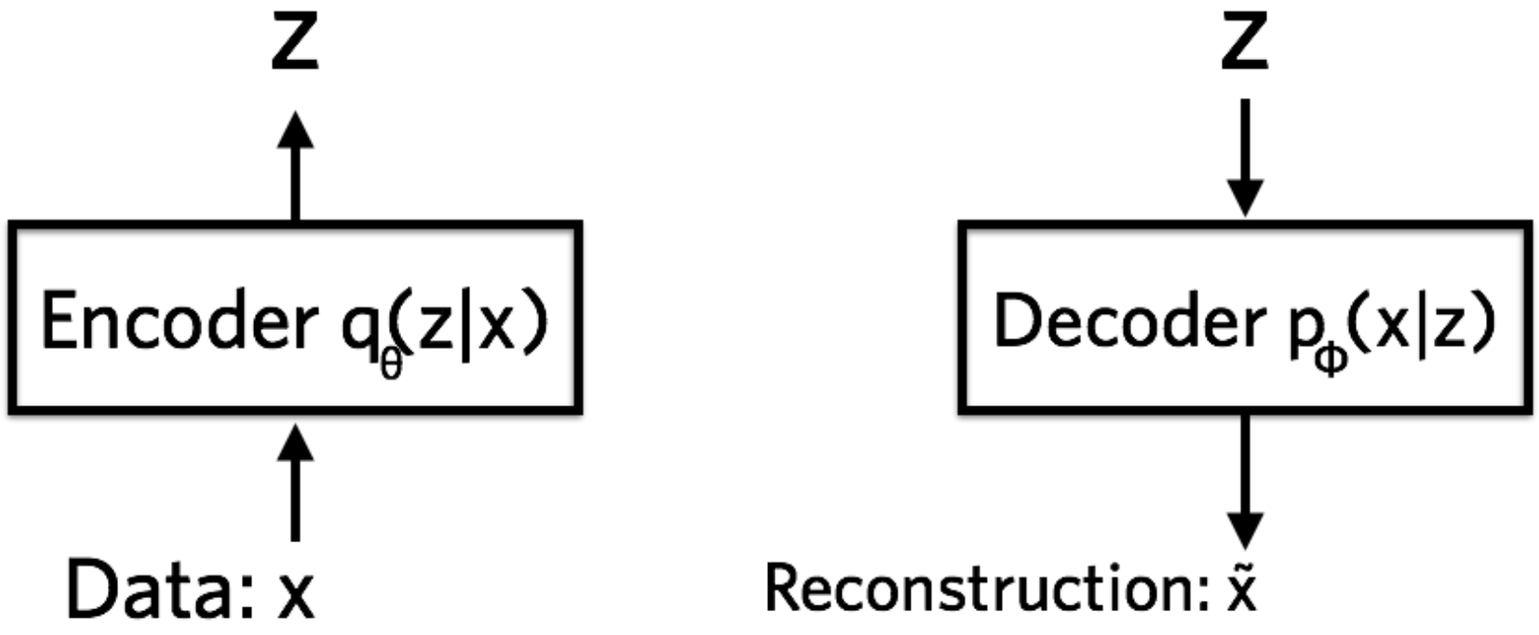
Diederik P. Kingma
Machine Learning Group
Universiteit van Amsterdam
dpkingma@gmail.com

Max Welling
Machine Learning Group
Universiteit van Amsterdam
welling.max@gmail.com

Abstract

How can we perform efficient inference and learning in directed probabilistic models, in the presence of continuous latent variables with intractable posterior distributions, and large datasets? We introduce a stochastic variational inference and learning algorithm that scales to large datasets and, under some mild differentiability conditions, even works in the intractable case. Our contributions is two-fold. First, we show that a reparameterization of the variational lower bound yields a lower bound estimator that can be straightforwardly optimized using standard stochastic gradient methods. Second, we show that for i.i.d. datasets with continuous latent variables per datapoint, posterior inference can be made especially efficient by fitting an approximate inference model (also called a recognition model) to the intractable posterior using the proposed lower bound estimator. Theoretical advantages are reflected in experimental results.

https://arxiv.org/abs/1312.6114



- a probabilistic encoder $q_\phi(z|x)$, approximating the true (but intractable) posterior distribution $p(z|x)$, and
- a generative decoder $p_\theta(x|z)$, which notably does not rely on any particular input x.

Both the encoder and decoder are artificial neural networks (i.e. hierarchical, highly nonlinear functions) with tunable parameters ϕ and θ, respectively.

Learning these conditional distributions is facilitated by enforcing a plausible mathematically-convenient prior over the latent variables, generally a standard spherical Gaussian: $z \sim \mathcal{N}(0, I)$.

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

http://blog.fastforwardlabs.com/2016/08/12/introducing-variational-autoencoders-in-prose-and.html

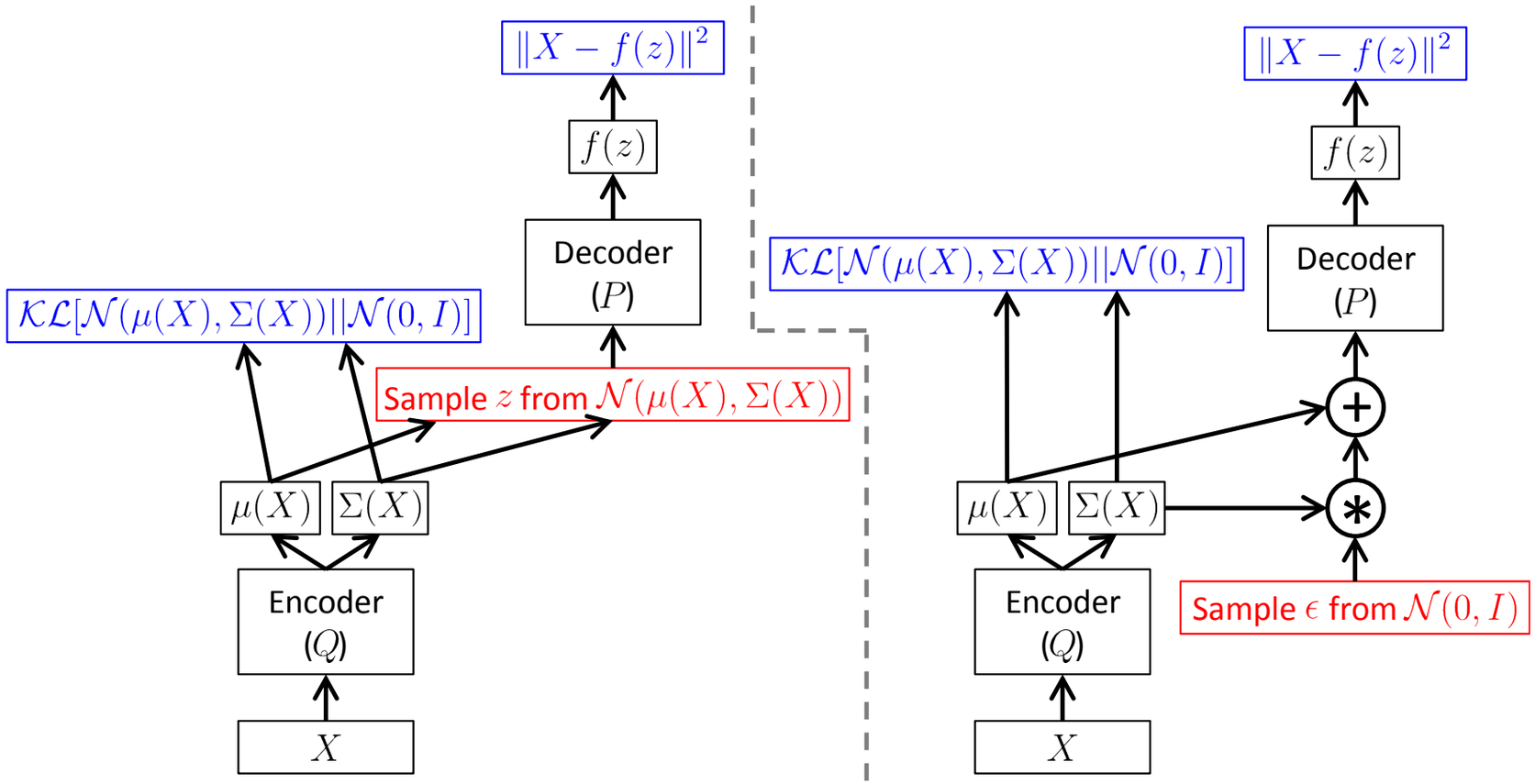
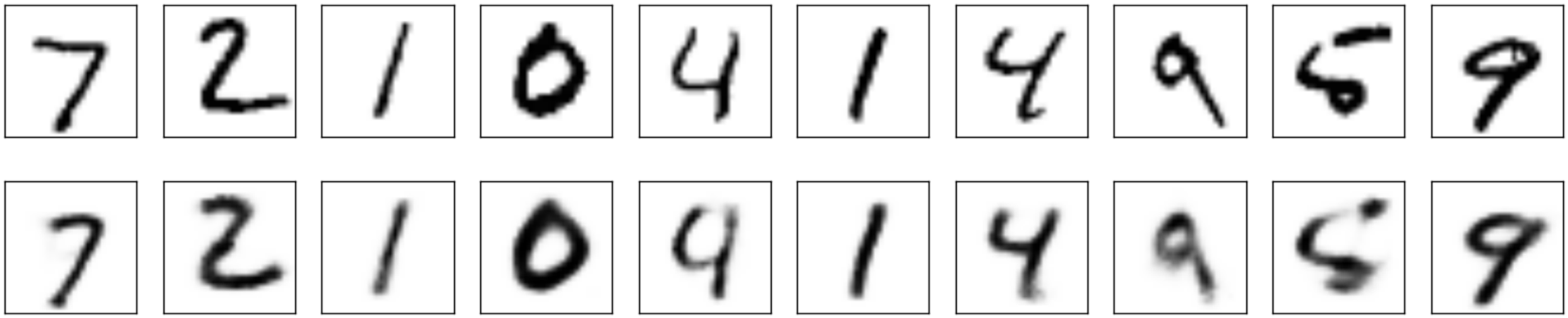
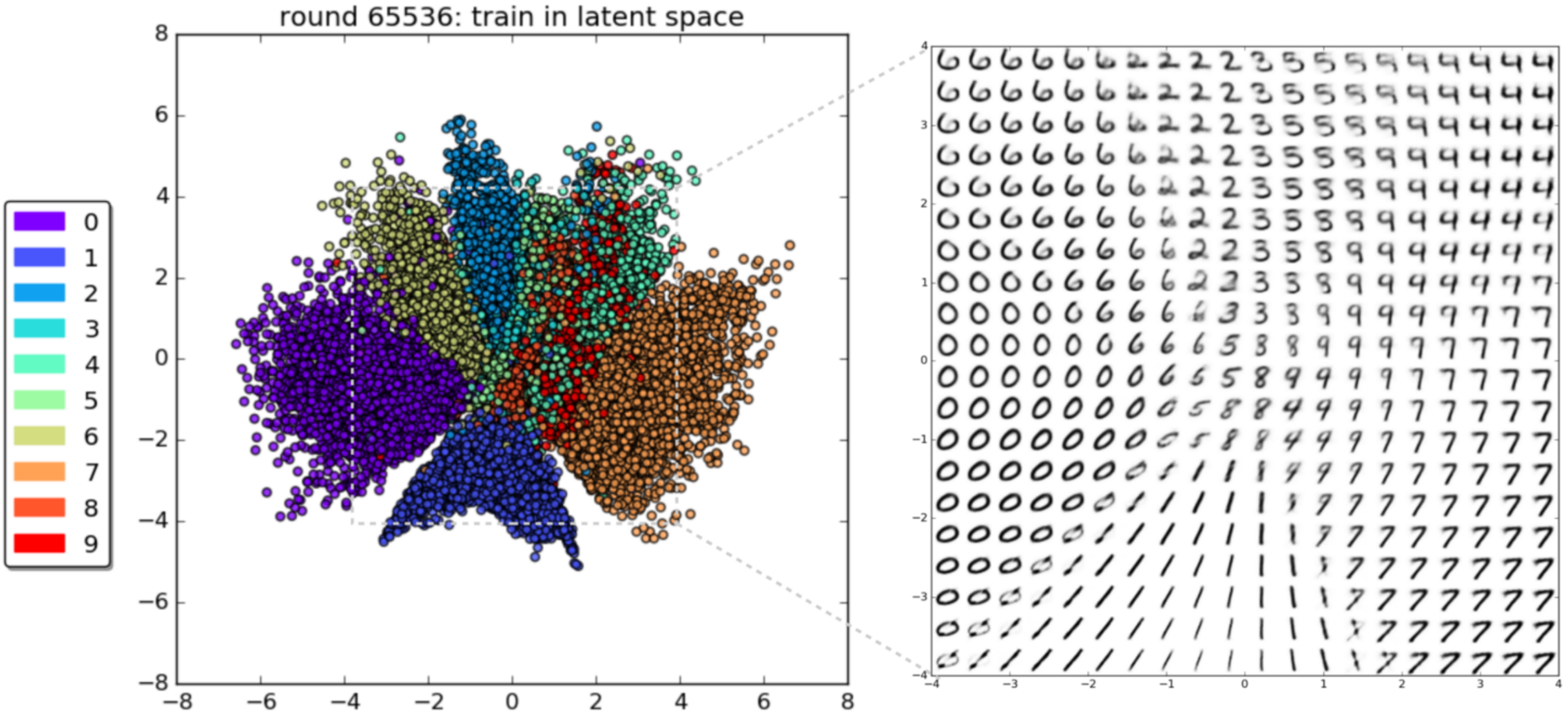


Figure 4: A training-time variational autoencoder implemented as a feedforward neural network, where $P(X|z)$ is Gaussian. Left is without the "reparameterization trick", and right is with it. Red shows sampling operations that are non-differentiable. Blue shows loss layers. The feedforward behavior of these networks is identical, but backpropagation can be applied only to the right network.

https://arxiv.org/pdf/1606.05908.pdf




round 65536: train in latent space
0
1
2
3
4
5
6
7
8
9

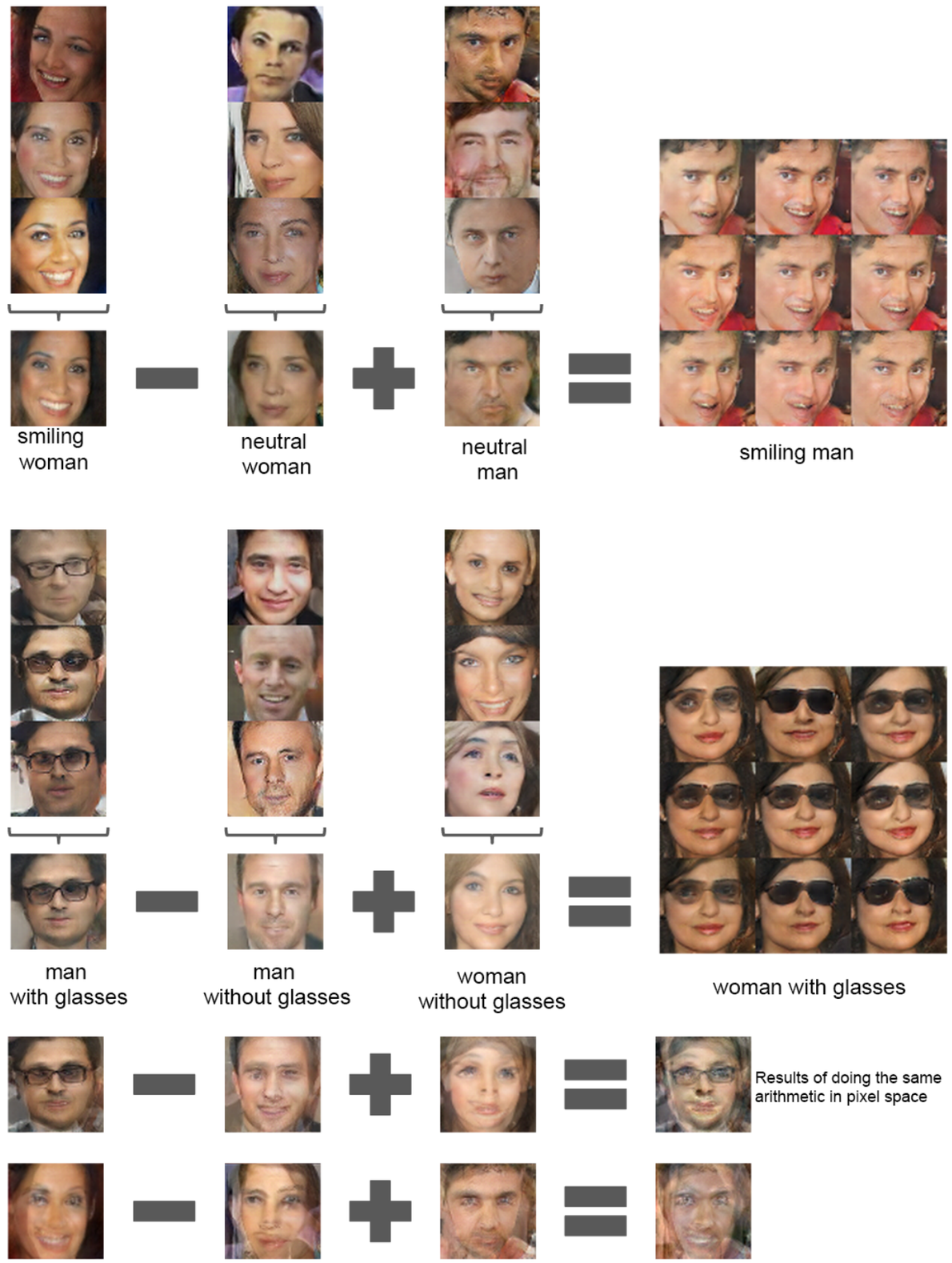


Figure 7: Vector arithmetic for visual concepts. For each column, the Z vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y. The center sample on the right hand side is produce by feeding Y as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform noise sampled with scale +-0.25 was added to Y to produce the 8 other samples. Applying arithmetic in the input space (bottom two examples) results in noisy overlap due to misalignment.

Figure 8: A ”turn” vector was created from four averaged samples of faces looking left vs looking right. By adding interpolations along this axis to random samples we were able to reliably transform their pose.

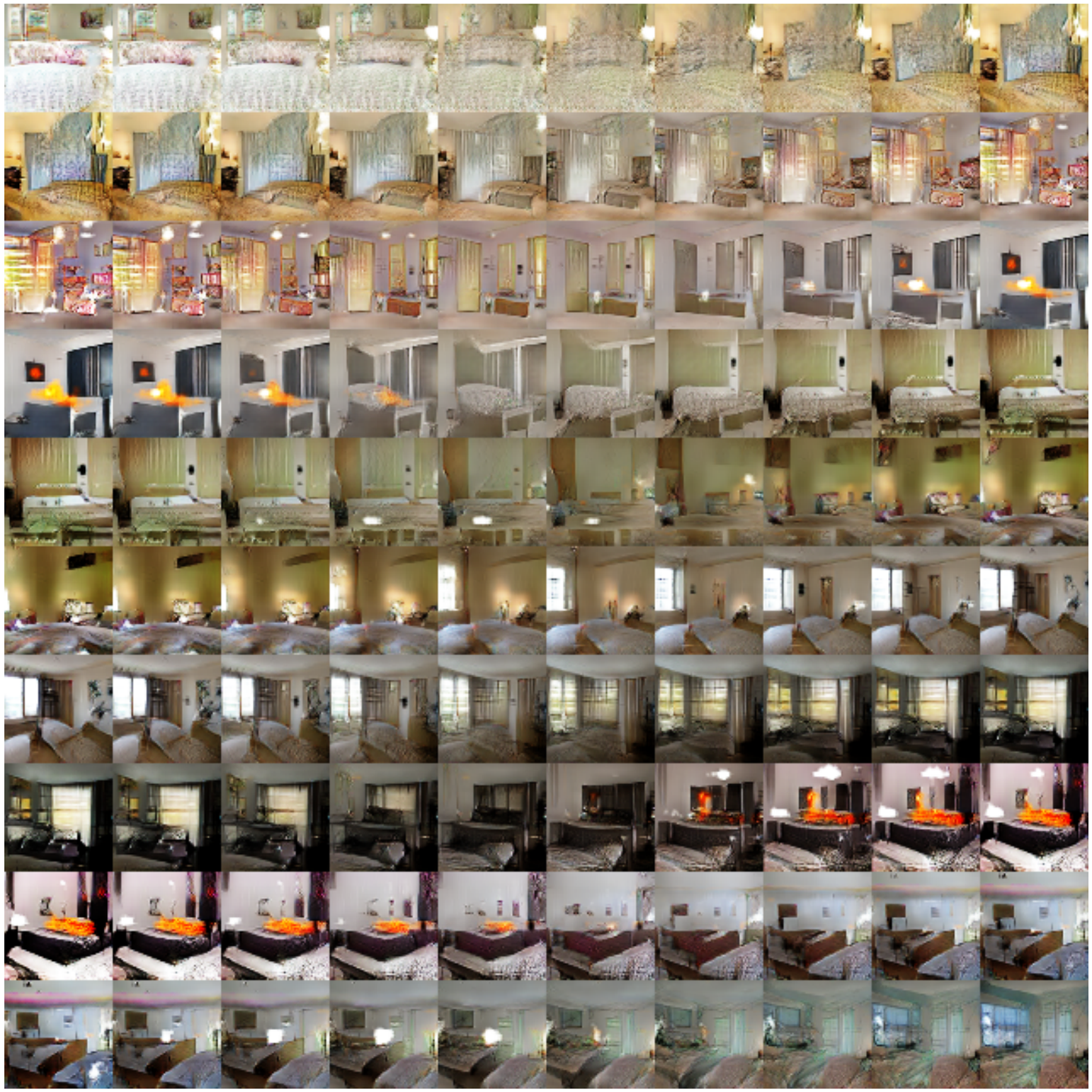


Figure 4: Top rows: Interpolation between a series of 9 random points in Z show that the space learned has smooth transitions, with every image in the space plausibly looking like a bedroom. In the 6th row, you see a room without a window slowly transforming into a room with a giant window. In the 10th row, you see what appears to be a TV slowly being transformed into a window.

https://arxiv.org/pdf/1511.06434v2.pdf

(a) Azimuth (pose)

(b) Elevation

(c) Lighting

(d) Wide or Narrow

Figure 3: **Manipulating latent codes on 3D Faces:** We show the effect of the learned continuous latent factors on the outputs as their values vary from -1 to 1. In (a), we show that one of the continuous latent codes consistently captures the azimuth of the face across different shapes; in (b), the continuous code captures elevation; in (c), the continuous code captures the orientation of lighting; and finally in (d), the continuous code learns to interpolate between wide and narrow faces while preserving other visual features. For each factor, we present the representation that most resembles prior supervised results [7] out of 5 random runs to provide direct comparison.

https://arxiv.org/pdf/1606.03657.pdf

GAN: Generative Adversarial Networks

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie,* Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair,† Aaron Courville, Yoshua Bengio‡
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

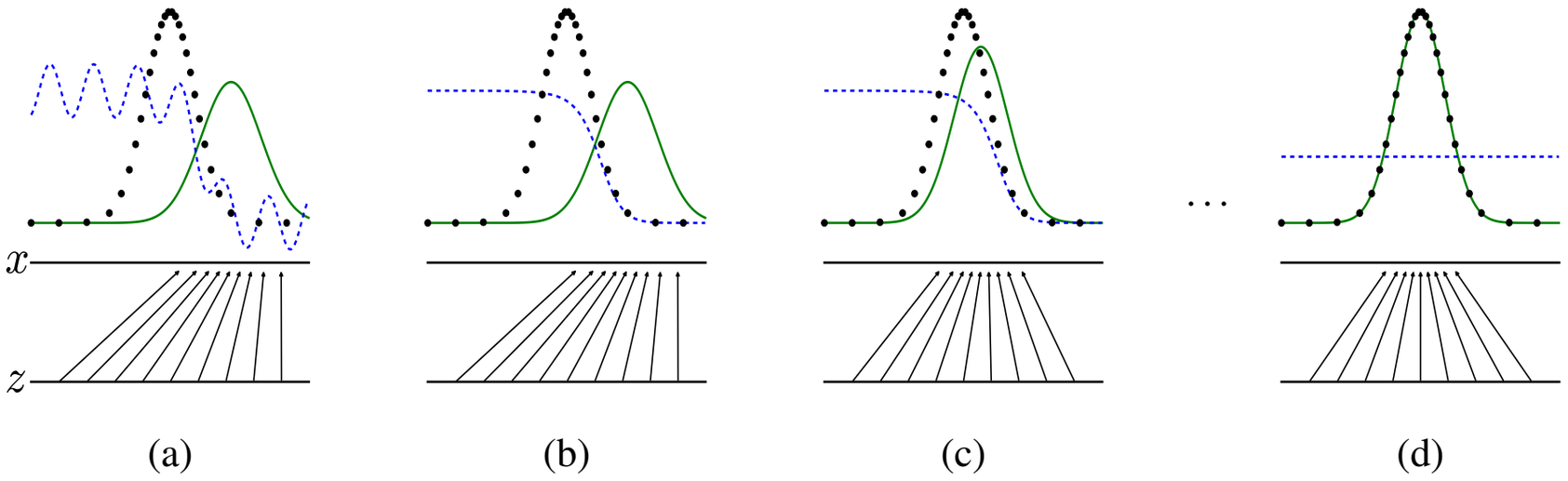
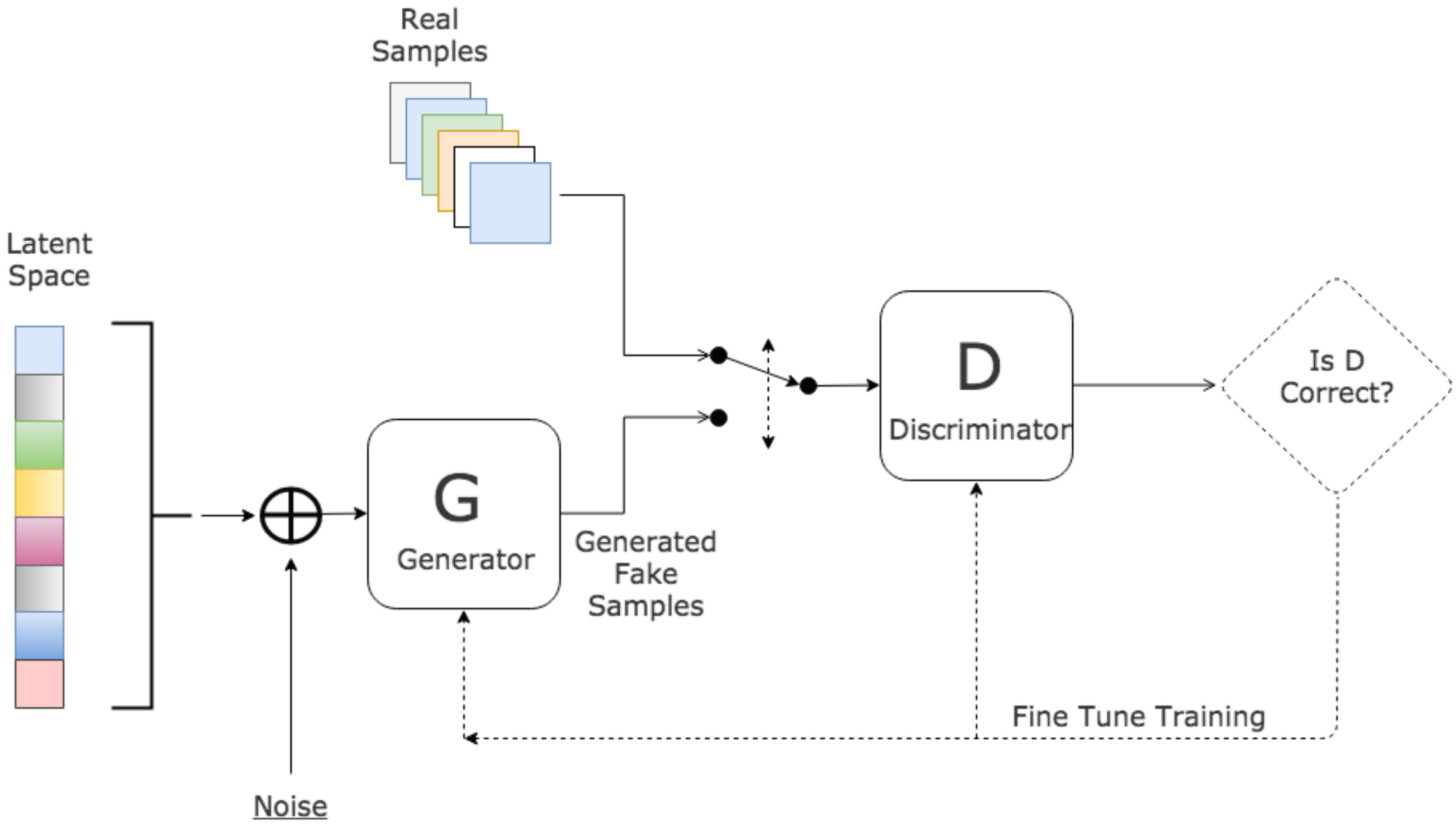


Figure 1: Generative adversarial nets are trained by simultaneously updating the **d**iscriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black, dotted line) $p_{\boldsymbol{x}}$ from those of the **g**enerative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which $\boldsymbol{z}$ is sampled, in this case uniformly. The horizontal line above is part of the domain of $\boldsymbol{x}$. The upward arrows show how the mapping $\boldsymbol{x} = G(\boldsymbol{z})$ imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_g(\boldsymbol{x})}$. (c) After an update to G, gradient of D has guided $G(\boldsymbol{z})$ to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$. The discriminator is unable to differentiate between the two distributions, i.e. $D(\boldsymbol{x}) = \frac{1}{2}$.

https://arxiv.org/abs/1406.2661

Example: http://cs.stanford.edu/people/karpathy/gan/

Generative Adversarial Network



http://www.kdnuggets.com/2017/01/generative-adversarial-networks-hot-topic-machine-learning.html

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 $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

  • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.

  • Update the discriminator by ascending its stochastic gradient:

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

  end for

  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

  • Update the generator by descending its stochastic gradient:

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

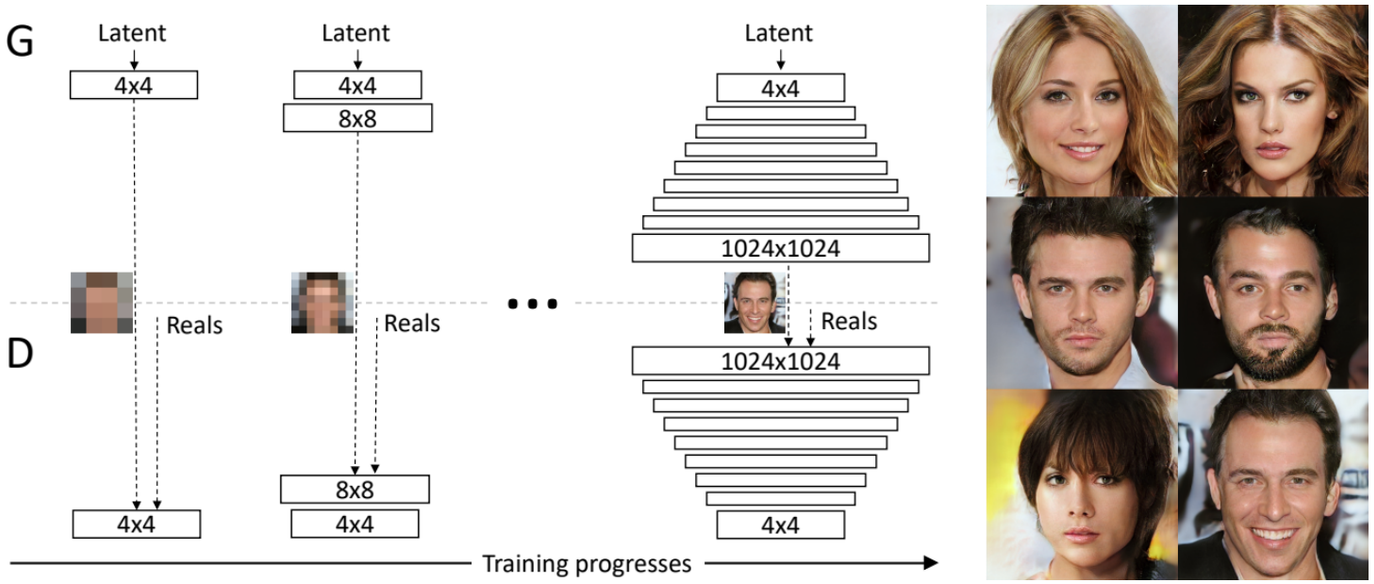
end for

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

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

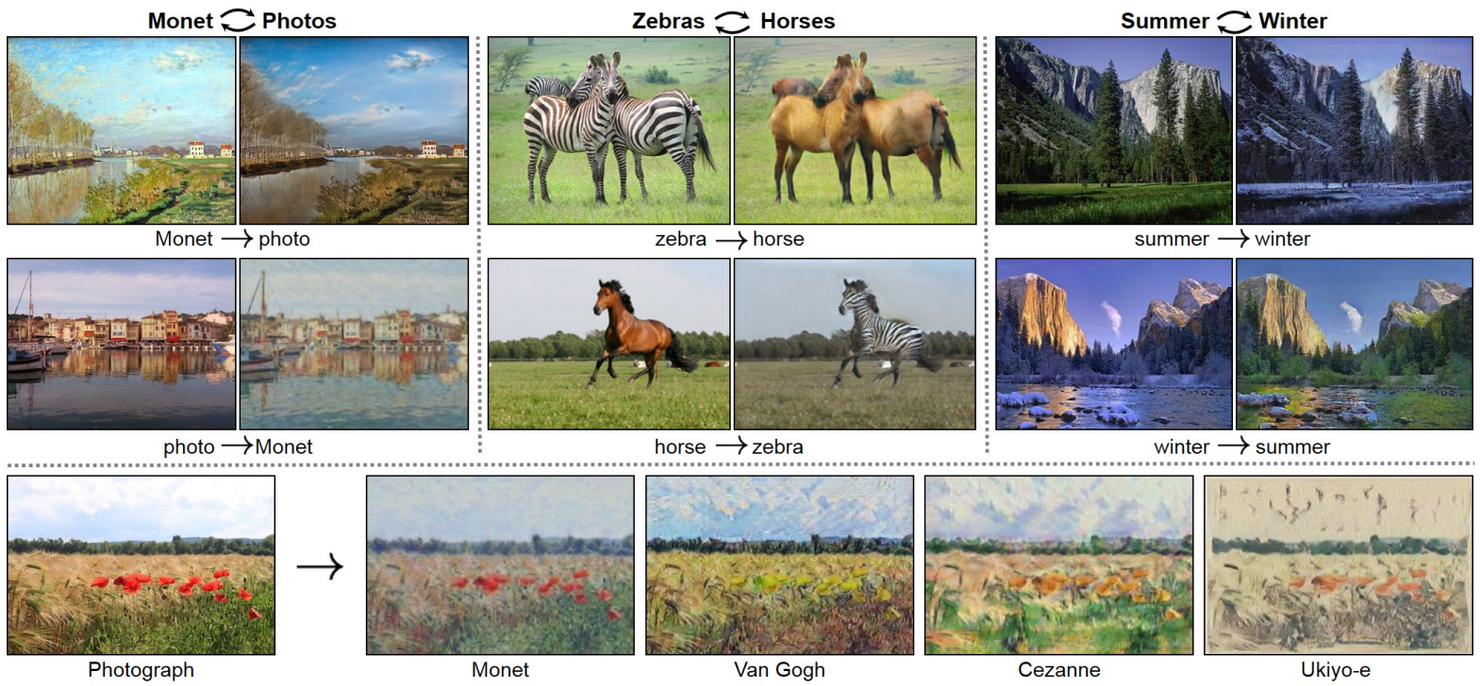
State-of-the-art

Karras et al, 2017, arXiv:1710.10196



Style transfer

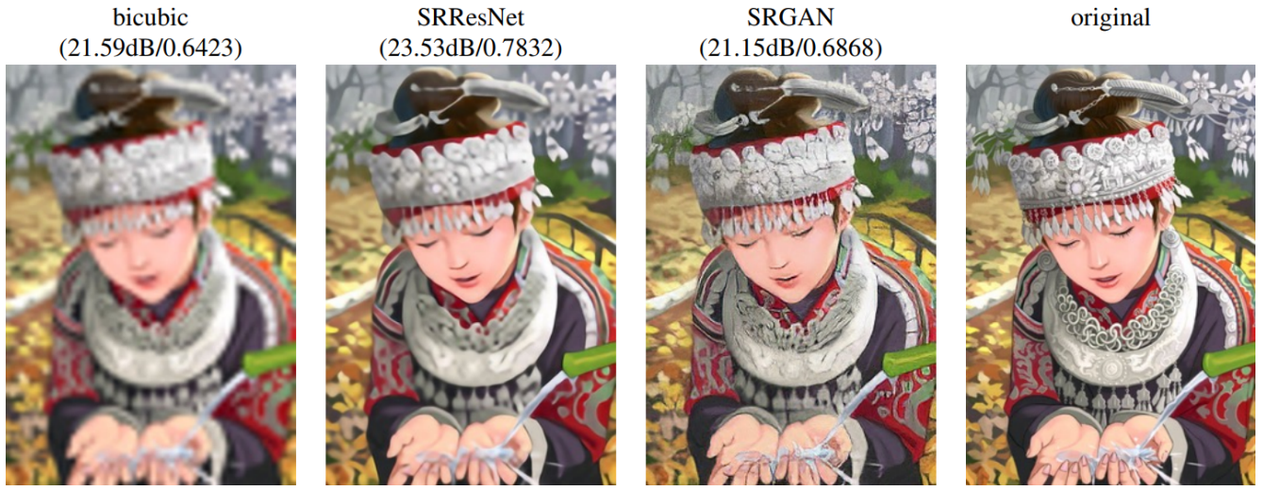
Zhu et al, 2017, arXiv:1703.10593



Slides from Gilles Louppe

Ledig et al, 2016, arXiv:1609.04802

Super-resolution



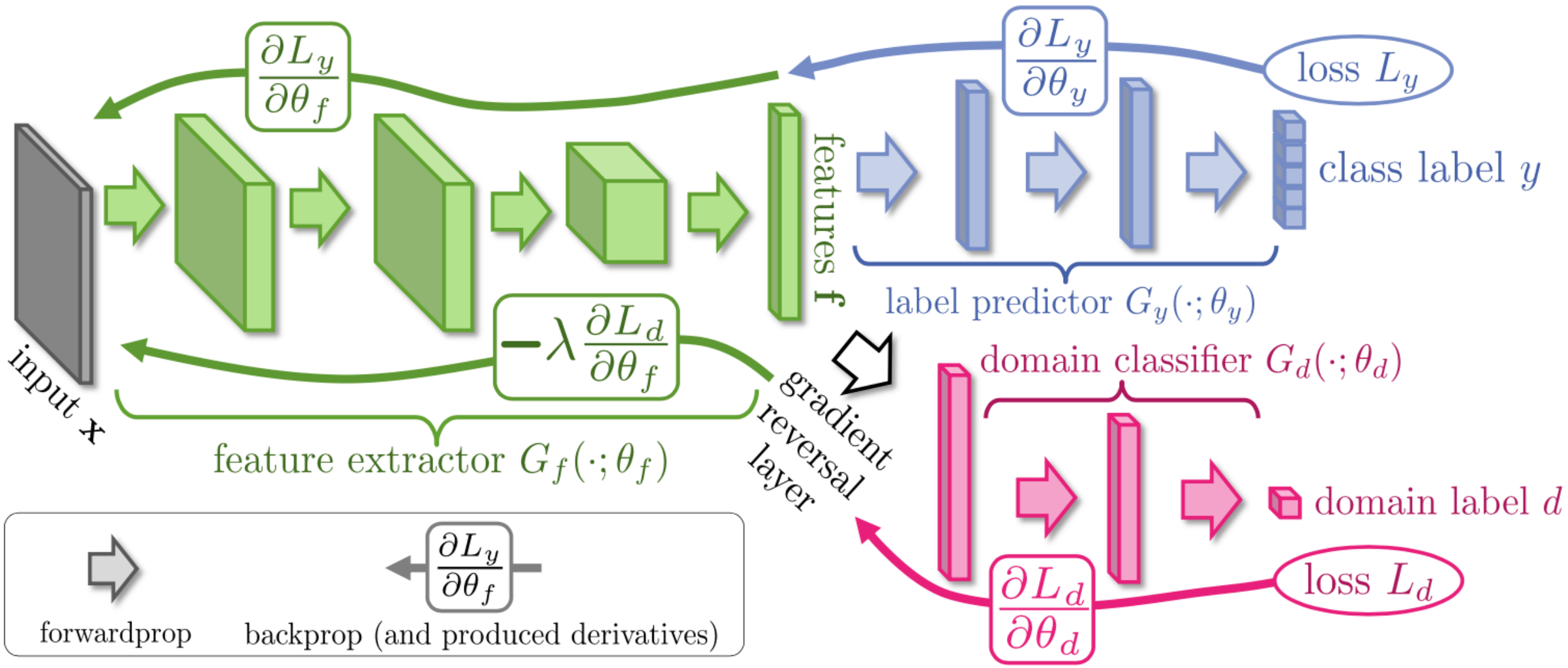
Zhang et al, 2016, arXiv:1612.03242

Text-to-image synthesis

Slides from Gilles Louppe

Adversarial

Domain Adaptation



Pivot

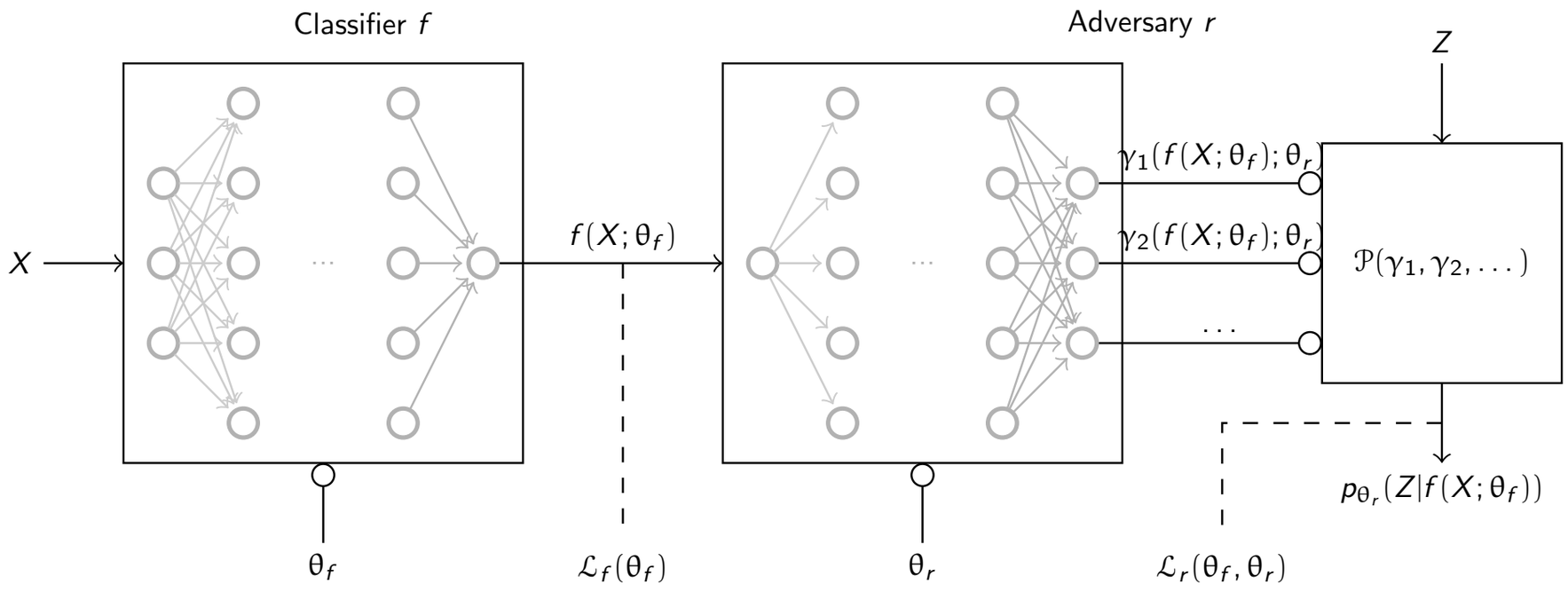
Learning to Pivot with Adversarial Networks

Gilles Louppe
New York University
g.louppe@nyu.edu

Michael Kagan
SLAC National Accelerator Laboratory
makagan@slac.stanford.edu

Kyle Cranmer
New York University
kyle.cranmer@nyu.edu

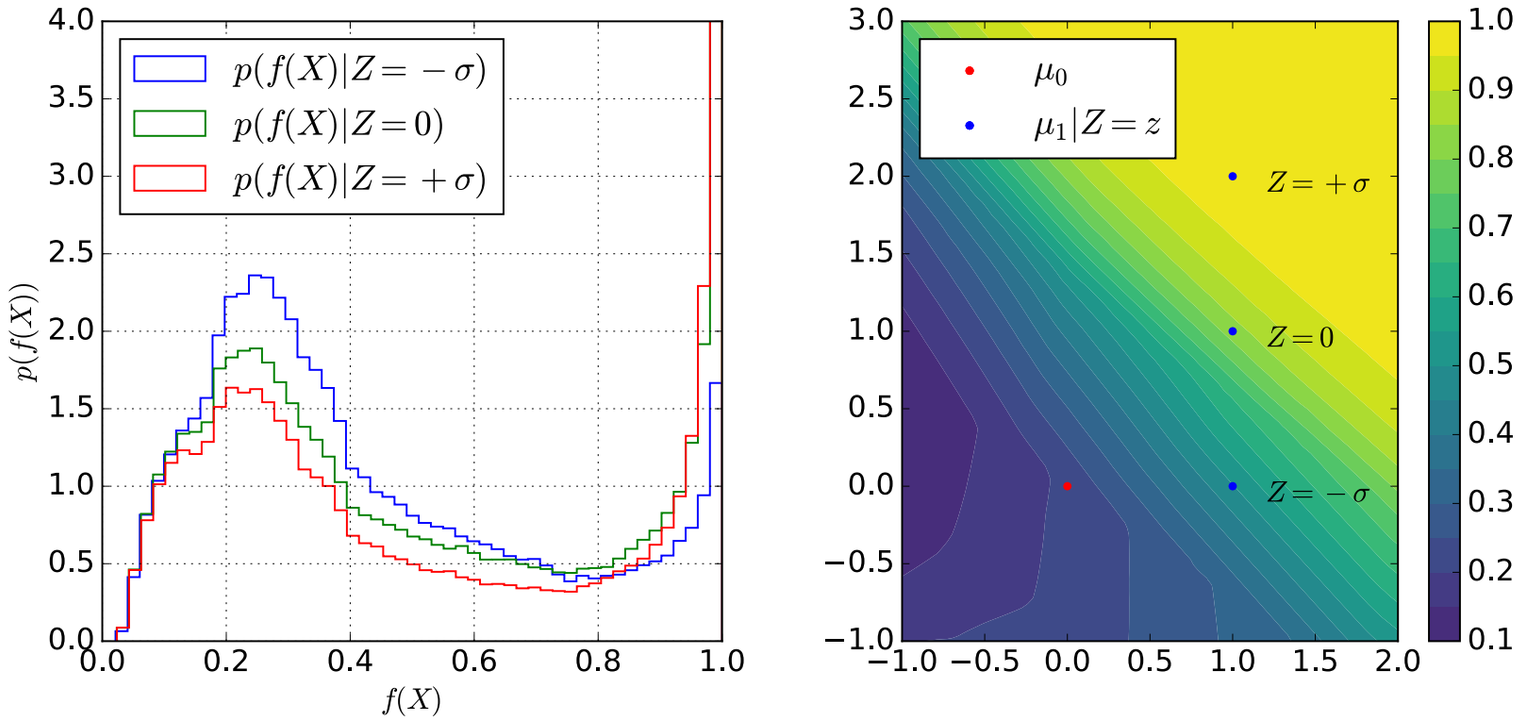
Adversarial game



Goal is to solve: $\hat{\theta}_f, \hat{\theta}_r = \arg\min_{\theta_f} \max_{\theta_r} \mathcal{L}_f(\theta_f) - \mathcal{L}_r(\theta_f, \theta_r)$

Intuitively, r penalizes f for outputs that can be used to infer Z.

https://arxiv.org/abs/1611.01046



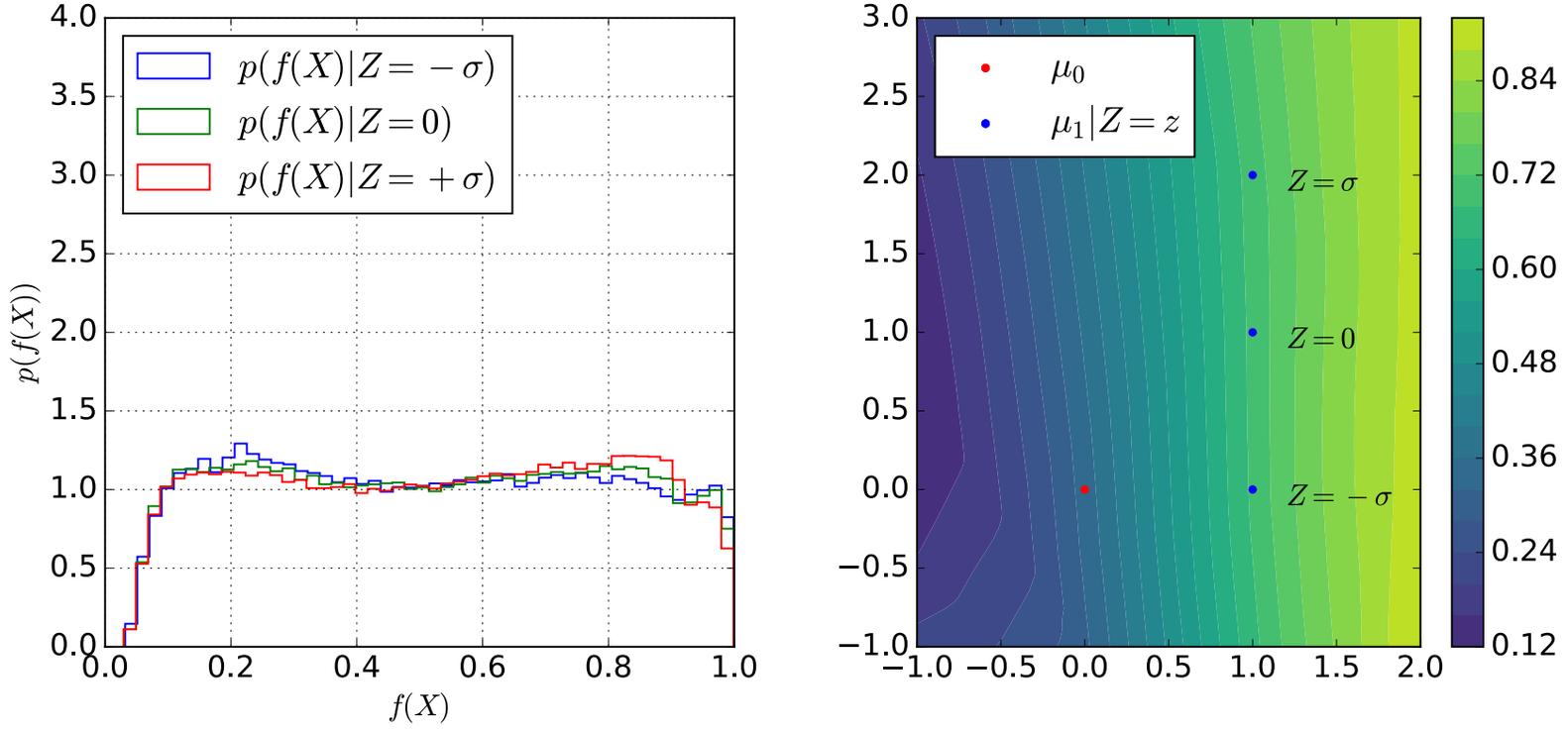


Figure 2: Toy example. (Left) Conditional probability densities of the decision scores at $Z = -\sigma, 0, \sigma$ without adversarial training. The resulting densities are dependent on the continuous parameter Z, indicating that f is not pivotal. (Middle left) The associated decision surface, highlighting the fact that samples are easier to classify for values of Z above σ, hence explaining the dependency. (Middle right) Conditional probability densities of the decision scores at $Z = -\sigma, 0, \sigma$ when f is built with adversarial training. The resulting densities are now almost identical to each other, indicating only a small dependency on Z. (Right) The associated decision surface, illustrating how adversarial training bends the decision function vertically to erase the dependency on Z.

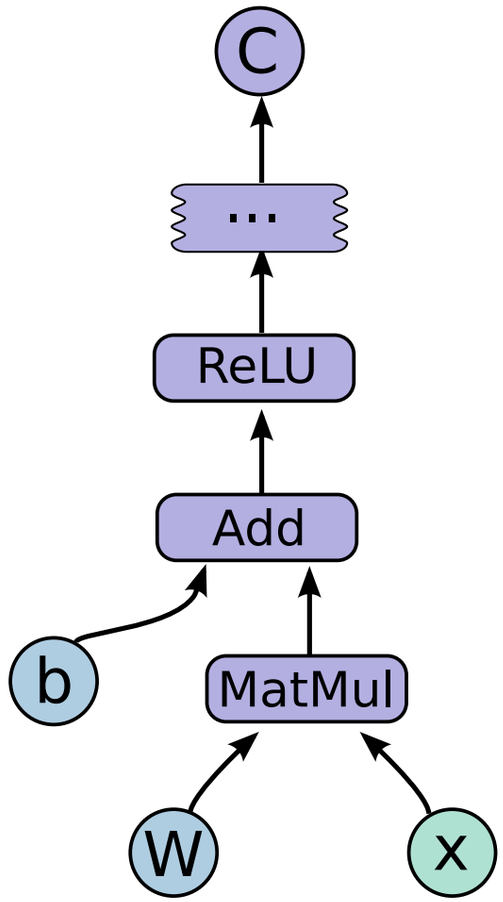
DL Software and Technical Challenges

numpy, TF, Keras

- Numpy
    - Provides a tensor representation.
    - It's interface has been adopted by everyone.
        - e.g. HDF5, Then, TensorFlow, … all have their own tensors.
        - You can use other tensors, for the most part interchangeably with numpy.
    - Provides extensive library of tensor operations.
        - D = A * B + C, immediately computes the product of A and B matrices, and then computes the sum with C.
- Tensorflow
    - Allows you write tensor expressions symbolically.
        - A * B + C is an expression.
    - Compiles the expression into fast executing code on CPU/GPU: F(A,B,C)
    - You apply the Compiled function to data get at a result.
        - D=F(A,B,C)
- Keras
    - Neutral Networks can be written as a Tensor mathematical expression.
    - Keras writes the expression for you.

DNN Software

- Basic steps
  - Prepare data
  - Build Model
  - Define Cost/Loss Function
  - Run training (most commonly Gradient Decent)
  - Assess performance.
  - Run lots of experiments…



- 2 Classes of DNN Software: (Both build everything at runtime)
  - Hep-Framework-Like: e.g. Torch, Caffe, …
    - C++ Layers (i.e. Algorithms) steered/configured via interpreted script:
  - General Computation Frameworks: Theano and TensorFlow
    - Everything build by building mathematical expression for Model, Loss, Training from primitive ops on Tensors
    - Symbolic derivatives for the Gradient Decent
    - Builds Directed Acyclic Graph of the computation, performs optimizations
    - Theano-based High-level tools make this look like HEP Frameworks (e.g. pylearn2, Lasagna, Keras, …)

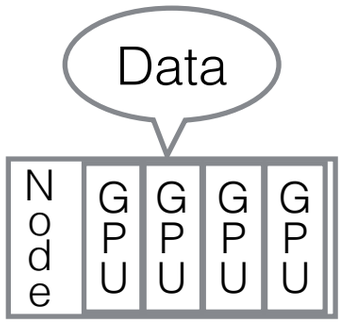
Technical Challenges

- Datasets are too large to fit in memory.
- Data comes as many files, sometimes organized in directories.
- For training, data needs to be read, mixed, “labeled”, possibly augmented, and normalized…. can be time consuming.
- Very difficult to keep the GPU fed with data. GPU utilization often < 10%, rarely > 50%.
- DL frameworks now mostly have tools that help with these tasks.

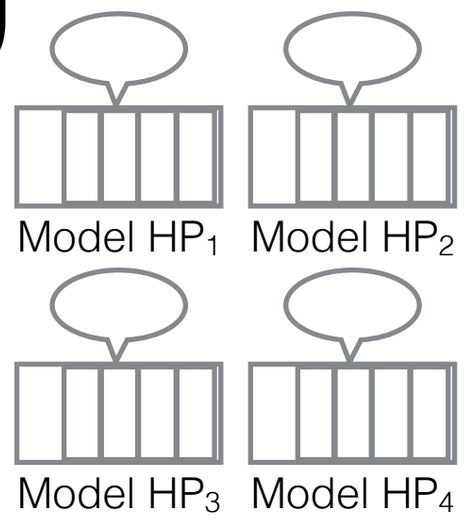
Distributed Training

1. *Tensor operation parallelism*: GPUs, FPGA, and ASICs (Google's Tensor Processing Unit).

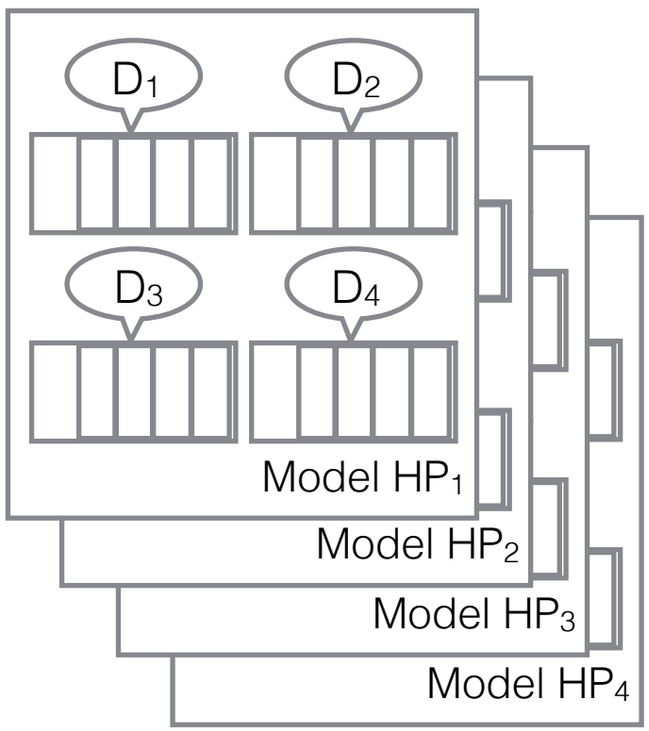
   - Note additional HN, Data, Model parallelism with multi-GPU



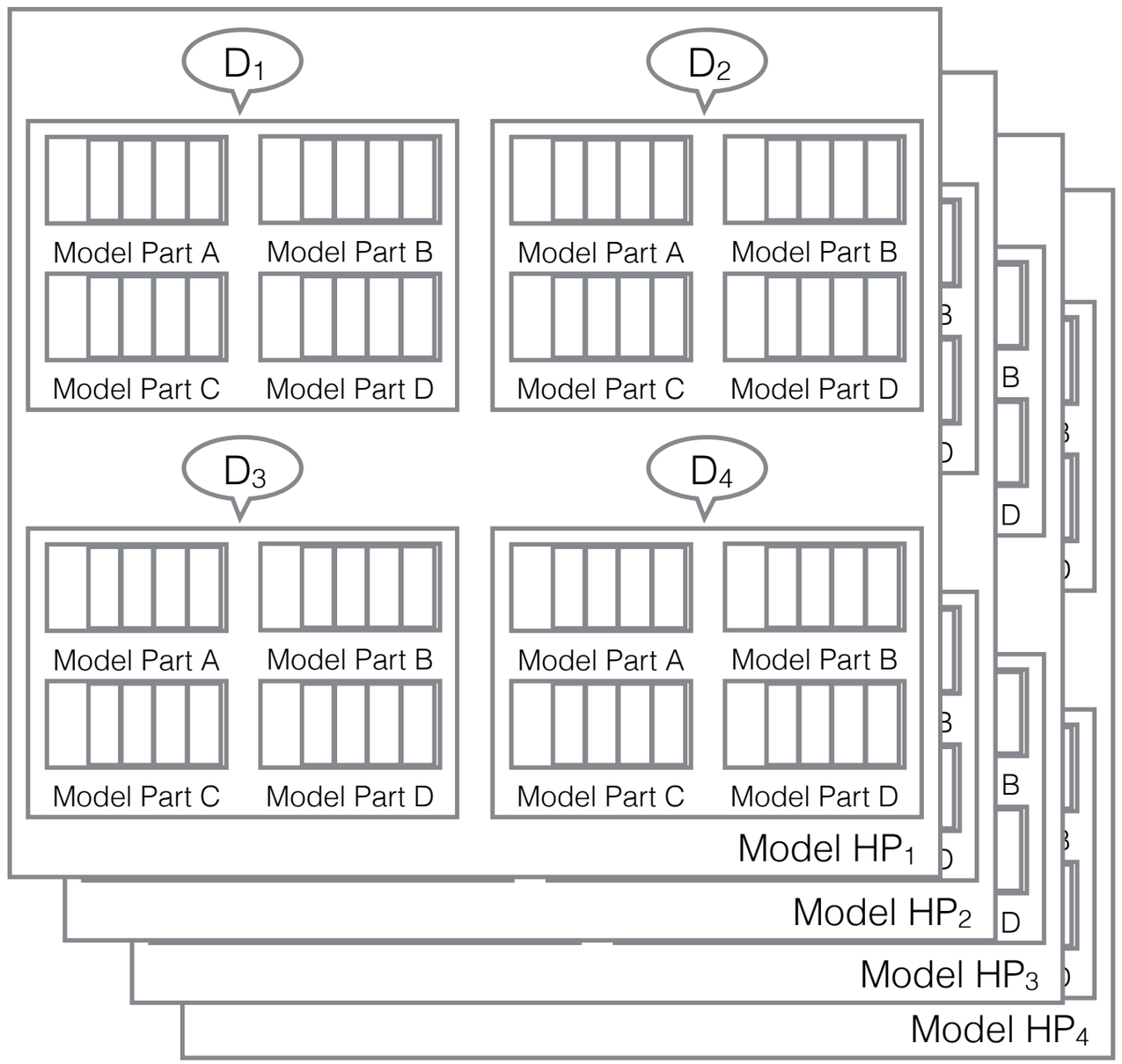
2. *Hyper-parameter scan*: simultaneously train multiple models. *e.g.* 1 model per GPU or node.



3. *Data Parallelism*: Each GPU or Node computes gradient on sub-set of data. Sync'ing gradients bottlenecked by bus or network.



4. *Model Parallelism*: Large model spread over many GPUs or nodes. Less network traffic but only efficient for large models.



Keras

https://keras.io/