Notes for Stat 991: Topics in Deep Learning

Edgar Dobriban*

May 14, 2019

Abstract

Deep learning has achieved many empirical successes, and has attracted considerable attention. However, it continues to be poorly understood. This advanced seminar course will explore several topics in deep learning. We will discuss both theory and applications.

Contents

1 Deep feedforward neural networks **3**
- 1.1 The model . . . 3
- 1.2 Training . . . 5
  - 1.2.1 Backpropagation . . . 5
  - 1.2.2 Regularization . . . 7
  - 1.2.3 Other optimization steps . . . 8
- 1.3 Notes on using DL . . . 9
- 1.4 Miscellanea . . . 14

2 Convolutional neural networks (CNNs) **19**
- 2.1 The problem and model . . . 19
- 2.2 Other aspects . . . 21
- 2.3 Training . . . 22
- 2.4 Graph CNN . . . 23
- 2.5 Shapes beyond images, invariance . . . 28
- 2.6 Spatial Transformer Networks . . . 30

3 Recurrent neural networks (RNNs) **31**
- 3.1 Setup . . . 31
- 3.2 Sequence Learning . . . 34
  - 3.2.1 Motivation . . . 34
- 3.3 Sequence Models . . . 37
  - 3.3.1 RNNs . . . 37

*Wharton Statistics Department, University of Pennsylvania. dobriban@wharton.upenn.edu. These notes draw inspiration from many sources, including David Donoho's course Stat 385 at Stanford, Andrew Ng's Deep Learning course on deeplearning.ai, David Silver's RL course, Tony Cai's reading group at Wharton. Disclaimer: the notes may contain factual and typographical errors. Thanks to several people who have provided parts of the notes, including Zongyu Dai, Georgios Kissas, Jane Lee, Barry Plunkett, Matteo Sordello, Yibo Yang, Bo Zhang, Yi Zhang, Carolina Zheng.

3.3.2 Training Challenges . . . 39
3.3.3 LSTM and GRU . . . 41
3.4 Machine Translation . . . 42
3.4.1 Attention . . . 45
3.4.2 Transformer Model . . . 46
3.5 Image + Text . . . 51

4 Unsupervised learning **52**
4.1 Setup . . . 52
4.2 PCA, AE, VAE . . . 53
4.3 Generative adversarial networks (GAN) . . . 57
4.4 Wasserstein GAN . . . 61
4.4.1 WGAN . . . 63
4.4.2 Gradient penalty . . . 65

5 Sequential decision-making: from bandits to deep reinforcement learning **67**
5.1 Intro . . . 67
5.2 Bandits . . . 69
5.2.1 Policies and regret analysis . . . 72
5.2.2 Comparing policies . . . 74
5.2.3 Adversarial bandits . . . 76
5.2.4 Contextual Bandits . . . 80
5.2.5 Software and Miscellanea . . . 84
5.3 Reinforcement learning . . . 84
5.3.1 Setup . . . 84
5.3.2 L1. Introduction . . . 84
5.3.3 Definitions . . . 86
5.3.4 Relation to Contextual Bandits . . . 88
5.3.5 Markov Decision Process . . . 89
5.3.6 Planning by dynamic programming . . . 93
5.3.7 Model-free prediction . . . 93
5.3.8 Model-free control . . . 94
5.3.9 Scaling up RL. Value function approximation . . . 95
5.3.10 Policy gradient . . . 96
5.3.11 Integrating learning and planning . . . 97
5.3.12 Hierarchical RL . . . 98
5.3.13 Task-agnostic RL . . . 102
5.4 Other topics . . . 103

6 Special topics **103**
6.1 Adversarial examples . . . 103
6.2 Neural ODEs . . . 109
6.3 Physics informed NNs (PINNs) . . . 110
6.4 Information bottleneck and invariance . . . 110
6.5 Gradient-based optimization . . . 112
6.6 Design of new architectures - gradient computations . . . 122
6.7 Distributed training . . . 124
6.7.1 Notes from Smola's class . . . 126
6.8 AutoML . . . 128
6.9 Biological plausibility . . . 136
6.10 Accessibility . . . 136

6.11 Explainability (and interpretability) . . . 137
6.12 Model compression . . . 139
6.13 Others . . . 140
6.13.1 List . . . 140
6.13.2 Privacy . . . 141
6.13.3 Applications . . . 141

7 Theory **141**
7.1 Why do we need theory? . . . 141
7.2 Computational complexity and learning . . . 142
7.3 Approximation theory . . . 145
7.4 Optimization . . . 146
7.5 Generalization . . . 147
7.6 Harmonic analysis . . . 148
7.7 Probabilistic ML . . . 149
7.8 Information geometry . . . 150
7.9 Random Matrix Theory . . . 150
7.10 Physics . . . 152
7.11 Geometry . . . 153

8 How to learn practical DL **153**

1 Deep feedforward neural networks

1.1 The model

1. *Supervised learning problem.* We have training data $x^{[i]}, y^{[i]}$, $i = 1, \ldots, m$, where x is a *feature vector*, and y is a *class label*.

   E.g., $x^{[i]}$: pictures, and $y^{[i]}$: labels (e.g., cat, dog, ..).

   We want to learn a *classifier* h to predict on test data. A classifier is a function of the feature vector such that $\hat{y} = h(x)$ is a good guess of the class label.

2. *Deep feedforward neural networks* (deep nets) consider a class of models composed of iterations of linear and coordinate-wise nonlinear maps. Aka "multilayer perceptron" (Rosenblatt, 1958). See Figure 1.

   For a data vector x:

   Construct a sequence of *activation* vectors, $a^{[0]}, a^{[1]}, a^{[2]}, \ldots, a^{[L-1]}$, consisting of input, hidden layers, and output, as follows

   (a) *Input*:

   $$a^{[0]} = x.$$

   (b) *Hidden layers*: The activations $a^{[1]}, a^{[2]}, \ldots, a^{[L-1]}$ are vectors constructed sequentially. Each is a vector, so $a_i^{[j]}$ is the i-th *node* (or *neuron* or *unit*) in j-th layer. At every layer, the activations are computed in the following way:

   i. First, a linear filtering is applied to the activations of the previous layer.

   $$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}.$$

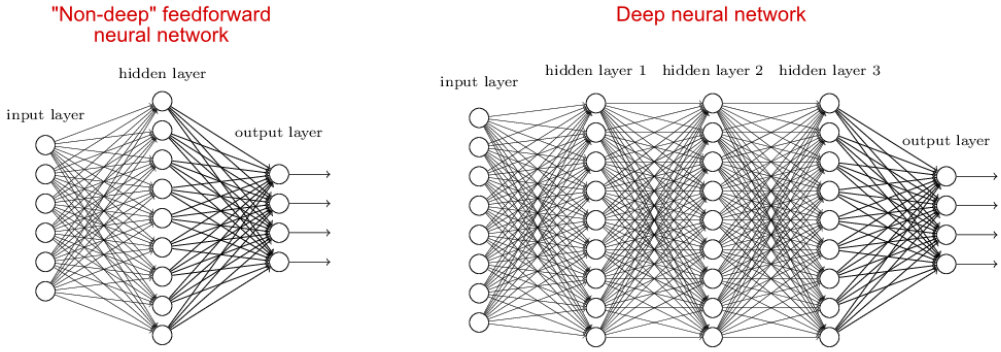


Figure 1: A shallow net and a deep net

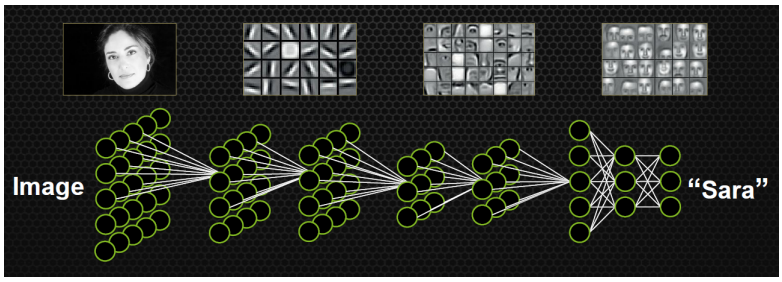


Figure 2: Example of feature learning.

This relies on the following parameters: $W^{[i]}$, $b^{[i]}$. For $i = 1, \ldots$. The *weights* W-s are matrices, while the *biases* b-s vectors.
The resulting vectors z are called *pre-activations*.
The dimension of the activation can increase or decrease (e.g., compression, denoising) every layer.

ii. Next, a nonlinear *activation function* σ is applied coordinate-wise to the pre-activation vectors z:

$$a^{[i]} = \sigma(z^{[i]}).$$

Popular examples:

- *rectified linear unit* (ReLU): $\sigma(x) = \max(x, 0)$.
- sigmoid: $\sigma(x) = 1/(1 + \exp(-x))$.

This can depend on i and then it is denoted as $\sigma^{[i]}$.

iii. Output layer:

$$\hat{y} = a^{[L]}.$$

For binary classification, the final layer is like a logistic regression.

3. Here are a few examples of deep learning visualizations to form intuition:

Andrej Karpathy's tools for "Deep Learning in your browser " https://cs.stanford.edu/people/karpathy/convnetjs/

Google's Neural Network Playground: http://playground.tensorflow.org

4. The intuition for neural networks is that they are able to extract *higher-level features* of the data (Figure 2). For instance
   (a) in the first layer, the weights can detect any linear combination of the input feature coordinates, (e.g., edges)
   (b) and then the activation function ensures that when there is a sufficient amount of that feature, it is propagated to the next layer.

   Moreover, the algorithms are thought to *automatically learn* the optimal weights for accurate classification. For instance in image classification, the lowest level features are often edge detectors, while the higher-level features look for combinations of edges in the right position, such as eyes in a face.

   The first examples of multilayer NNs, such as Fukushima's neurocognitron (1980) were tuned heuristically, without learning.

   Before NNs, the state of the art in computer vision was to use any of a large number of libraries of features, combined with a linear . Still very complicated, still heuristic.

5. Next, let us see how we can express the model with many training examples. We will denote the training examples as $x^{[i]}$. We will also write for $a^{[j](i)}$ the i-th node corresponding to the j-th example.
   (a) *Input layer*:

   $$A^{[0]} = X.$$

   Here X a matrix that has the training data as its columns.
   Matrix dimension: $n_x \times m$, where n_x is dimension of samples (usually denoted by p in statistics), and m is number of samples (usually denoted by n in statistics).
   (b) "*Forward propagation*":
   Activations $A^{[i-1]}$ arranged in a matrix.
   Matrices are organized as follows: columns - training examples/filters, rows - units/nodes.

   $$Z^{[i]} = W^{[i]} A^{[i-1]} + b^{[i]},$$

   $$A^{[i]} = \sigma(Z^{[i]})$$

   (c) *Output layer*:

   $$\hat{y} = A^{[L]}.$$

   There can be multiple outputs (e.g., center of car, width and height of the box enclosing it)

1.2 Training

1.2.1 Backpropagation

1. *Loss function*: Consider only binary classification for simplicity. For one training example, we will use the *logistic loss*:

   $$\ell(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

e.g., if $y = 1$, loss is $\log(1/\hat{y})$ [plot]

For multiple training examples, consider the *empirical risk*:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}^{[i]}, y^{[i]})$$

Here $\hat{y}$ are predictions of the neural network (defined above) with parameters $W = (W^{[1]}, \ldots)$ and $b = (b^{[1]}, \ldots)$.

2. Training proceeds via *gradient descent* (GD), or versions thereof.

   Fix some initialization of the parameters W, b, and a *learning rate* α. The parameters will be updated iteratively as

$$W := W - \alpha\, dW$$

$$b := b - \alpha\, db$$

   The convention is that for parameters θ we denote the Jacobian $d\theta := \partial J/\partial\theta$.

   Technically, the loss may not be diffferentiable everywhere, if we use non-smooth non-linearities such as ReLU. It turns out, that the probability of this being a problem is virtually non-existent with proper random initialization.

3. We will show the form of GD for one data sample x. In practice, one would do it for a small randomly chosen data sample (e.g., 256 images). This is known as *mini-batch stochastic gradient descent* (SGD). One would vectorize the results as much as possible, for speed.

   We want to compute the derivatives of the empirical risk with respect to the weights $W^{[i]}$ and biases $b^{[i]}$.

   The computation proceeds backwards, starting from $dW^{[L]}$, and using the chain rule to compute $dW^{[i-1]}$ in terms of $dW^{[i]}$. Known as "*backpropagation*", (Rumelhart et al, 1986). Invented independently in several communities (e.g., "adjoint state method" in optimal control) and by several groups of researchers in neural networks (e.g, Sejnowski, LeCun) [Draw *computation graph*]

   (a) The last layer is special, because of the loss, and due to the use of the sigmoid:

$$dW^{[L]} = \frac{\partial \ell(\hat{y}, y)}{\partial W^{[L]}} = \frac{\partial \ell(\hat{y}, y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^{[L]}}$$

   Now

$$\frac{\partial \ell(\hat{y}, y)}{\partial \hat{y}} = -y \frac{\log(\hat{y})}{\partial \hat{y}} - (1 - y) \frac{\log(1 - \hat{y})}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{(1 - y)}{1 - \hat{y}}$$

   Also,

$$\frac{\partial \hat{y}}{\partial W^{[L]}} = \frac{\partial \sigma(z^{[L]})}{\partial W^{[L]}} = \frac{\partial \sigma(z^{[L]})}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial W^{[L]}}$$

   At the last layer, we use sigmoid $\sigma(x) = 1/(1+\exp(-x))$, with $\sigma'(x) = \sigma(x)(1-\sigma(x))$, so that

$$dz^{[L]} = \frac{\partial \sigma(z^{[L]})}{\partial z^{[L]}} = \sigma(z^{[L]})(1 - \sigma(z^{[L]})) = \hat{y}(1 - \hat{y})$$

   From the two equations above we find the simplified form

$$\frac{\partial \ell(\hat{y}, y)}{\partial z^{[L]}} = \hat{y} - y$$

Finally

$$\frac{\partial z^{[L]}}{\partial W^{[L]}} = \frac{\partial (W^{[L]} a^{[L-1]} + b^{[L]})}{\partial W^{[L]}} = a^{[L-1]}$$

We also have, for the bias term,

$$db^{[L]} = \frac{\partial \ell(\hat{y}, y)}{\partial b^{[L]}} = \frac{\partial \ell(\hat{y}, y)}{\partial z^{[L]}} \frac{\partial z^{[L]}}{\partial b^{[L]}}$$

The first term was already computed, while the last term is the identity.
This shows how to do backpropagation for the last layer.

(b) For the remaining layers, the calculation of $da^{[i-1]}, dz^{[i-1]}, \ldots$ is similar (and also the same calculation for all layers), assuming now that the derivatives $da^{[i]}, dz^{[i]}, \ldots$ for higher layers have been computed:

$$da^{[i-1]} = \frac{\partial \ell(\hat{y}, y)}{\partial a^{[i-1]}} = \frac{\partial \ell(\hat{y}, y)}{\partial z^{[i]}} \frac{\partial z^{[i]}}{\partial a^{[i-1]}} = dz^{[i]} W^{[i]}$$

Also

$$dz^{[i-1]} = da^{[i-1]} \frac{\partial z^{[i-1]}}{\partial a^{[i-1]}} = da^{[i-1]} / \sigma'(a^{[i-1]})$$

$$dW^{[i-1]} = dz^{[i-1]} a^{[i-1]}$$

and

$$db^{[i-1]} = dz^{[i-1]}.$$

This shows how to do backpropagation for intermediate layers.

4. The *learning rate* hyperparameter α is chosen as a small value, such as $\alpha = 0.01$.
5. See also http://colah.github.io/posts/2015-08-Backprop/

1.2.2 Regularization

1. The number of weights can be much larger than the sample size. Regularization is used to avoid overfitting.
2. An ℓ_2 *penalty* is often used for regularization. Minimize

$$J(W, b) + \frac{\lambda}{2} \sum_{i=1}^{L} \|W^{[i]}\|_F^2$$

Also known as *weight decay*, because the implementation reduces to

$$W^{[i]} := (1 - \alpha/\lambda) W^{[i]} - \alpha \, dW^{[i]}$$

3. *Dropout*: Randomly zero out units, to mitigate weight *co-adaptation* (Srivastava et al 2014).

At train time: Draw masks for the activations with iid Bernoulli entries $d^{[i]}$, say with success probability $q = 0.8$:

$$d_j^{[i]} \sim_{iid} Bernoulli(q)$$

Zero out units:

$$a^{[i]} = a^{[i]} \odot d^{[i]}$$

Rescale, aka "*inverted dropout*", in order to "not reduce the expected value of the next activation":

$$a^{[i]} = a^{[i]}/q$$

Then keep the same iterations. This corresponds to freezing some randomly chosen weights at each iteration.

At test time: change nothing.

Note: The original dropout applies rescaling at test time instead.

4. *Data augmentation*: Increase the sample size, by adding transforms of the training samples to the dataset. E.g., for images: random shift, zoom, small rotation, L-R flip, Random cropping, Shearing, Color shifting.

1.2.3 Other optimization steps

1. *Input normalization.* Normalize each feature of the input data to have unit norm. Reduces condition number of objective.
2. Initialization: typically *random initialization* is used. For instance, iid Gaussian weights, with some small variance.

$$W^{[l]} \sim \mathcal{N}(0, \sigma_l^2 \cdot I_{n_l} \otimes I_{n_{l-1}}).$$

Here σ_l^2 is some function of the dimension and activation. For instance

$$\sigma_l^2 = 1/n_{l-1}$$

for linear and tanh (Xavier initialization). Also,

$$\sigma_l^2 = 2/n_{l-1}$$

for ReLU. These can work better than others such as $\sigma_l^2 = 1/(n_l + n_{l-1})$.

3. *Residual networks (ResNets)*, (He et al, 2015). Allow the information from a layer to "skip" a node. Empirically, allow the training of much deeper networks.

Recall that two layers of a feedforward network have the form

$$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}.$$

$$a^{[l+1]} = \sigma(z^{[l+1]}).$$

$$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}.$$

$$a^{[l+2]} = \sigma(z^{[l+2]}).$$

A residual network allows the information from a layer to "skip" a node:

Name	Update Rule
SGD	$\Delta\theta_t = -\alpha g_t$
Momentum	$m_t = \gamma m_{t-1} + (1-\gamma) g_t,$ $\Delta\theta_t = -\alpha\, m_t$
Adagrad	$G_t = G_{t-1} + g_t^2,$ $\Delta\theta_t = -\alpha\, g_t G_t^{-1/2}$
Adadelta	$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$ $\Delta\theta_t = -\alpha g_t v_t^{-1/2} D_{t-1}^{1/2},$ $D_t = \beta_1 D_{t-1} + (1-\beta_1)(\Delta\theta_t/\alpha)^2$
RMSprop	$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$ $\Delta\theta_t = -\alpha\, g_t v_t^{-1/2}$
Adam	$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t,$ $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$ $\hat{m}_t = m_t/(1-\beta_1^t),$ $\hat{v}_t = v_t/(1-\beta_2^t),$ $\Delta\theta_t = -\alpha\, \hat{m}_t \hat{v}_t^{-1/2}$

Figure 3: Optimization methods.

$$a^{[l+2]} = \sigma(z^{[l+2]} + a^{[l]}).$$

The problem is that a linear transform $W^{[l+2]}a^{[l+1]}$ easily changes the "magnitude" of the activations. By adding the previous activations, we are now *learning the residual*, and the baseline is the identity. This is easier.

Training is very similar to before.

4. Batch normalization (Ioffe & Szegedy, 2015): Normalize the input to each layer to "prevent internal covariate shift".
5. Beyond SGD: There are several variants to accelerate SGD by adding a scaled difference of the previous steps: Adam (Kingma & Ba, 2015), RMSProp (Hinton, 2012). See overview paper by Ruder. See Figure 3.
6. Scaling the input to be in a reasonable range, such as $[0, 1]$.
7. Gradient clipping, elementwise.

1.3 Notes on using DL

1. **Flexibility.**

   The input-output x, y can be anything. This "*black-box*" thinking gives a lot of flexibility to practitioners. The art is to reduce everything to supervised learning. How do you solve a problem using deep learning? You reduce it to x, y pairs where x is input, y is output, and y depends in a "smooth" way on x. *Differentiable programming.*

   Examples:

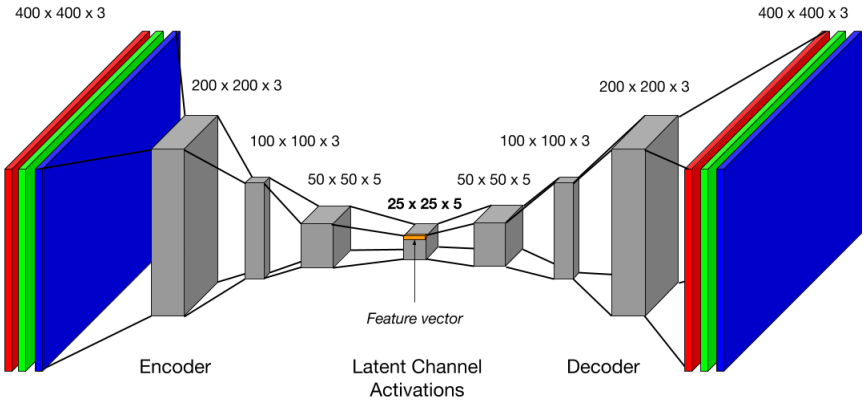


Figure 4: Autoencoder.

(a) $y = x$. Known as an *autoencoder* (Figure 4). This is an example of *unsupervised learning*. Denoises and compresses the input x if we constrain the intermediate layers.

(b) *Object localization*: Input x is an image. Output $y = (y_{center}, y_{height}, y_{width})$ is the position of a *bounding box* containing an object (such as a car) in the image. Can also detect multiple objects.

(c) Can transform data into images to make the existing techniques applicable (highly creative). For instance:

- Fraud detection: Mouse movement $\rightarrow$ image of a line.
- Genomics: Reads from a genome $\rightarrow$ image of colored dots.

2. **What is so special about deep net architecture?**

Purported special properties:

(a) *Universal approximators.* Can capture any structure, approximate any function. Deep > shallow?

(b) *Modular structure.* Can design architectures for any application, combining and reusing existing modules

(c) *Large capacity.* Performance keeps increasing with more data, doesn't saturate

(d) *No bias-variance tradeoff.* Bigger network reduces bias (also trains longer), *but* more data reduces variance. So, there is no tradeoff.

3. **What else do we need for this to work?** Data and compute.

Many research groups have reported good performance of deep neural networks for various problems. For instance the recent resurgence of interest in neural networks can be attributed to some extent to the 2012 paper by Krizhevsky, Sutskever, and Hinton, which was able to achieve good multi-class classification on the ImageNet dataset, with a specific type of convolutional neural network (CNN).

The same type of CNN architecture had already been used since the 1980s, by many others including Hinton himself. The new success was mainly due to much larger computational resources available (including training on graphical processing units or GPUs), and much larger data sets available. The ImageNet dataset has millions of high quality images. (See https://www.kaggle.com/c/imagenet-object-localization-challenge)

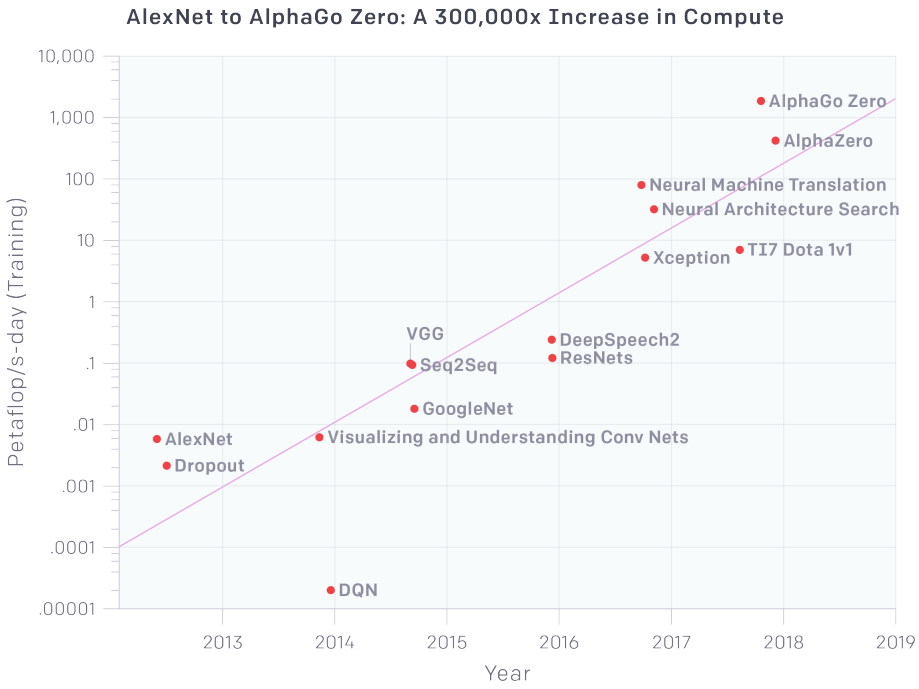


Figure 5: AI and compute. https://blog.openai.com/ai-and-compute/.

> It has only recently become possible to collect *labeled datasets with millions of images*"
> "current *GPUs*, paired with a highly-optimized implementation of 2D convolution, are powerful enough to facilitate the training of interestingly-large CNNs"

OpenAI, Figure 5:

> Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.5 month-doubling time.

4. **Economic value.**
   NNs are also widely used for *online advertising*. Specifically they are used to predict which ad to display to users to maximize expected profits. It is thought that most of the short-term *economic value* that they generate is from this area. This may not be a surprise given that a large fraction of the revenue of internet companies like Google comes from online ads.
5. **Concerns in the community, and failures of deep learning.**
   (a) *Self-driving car accident.* Tesla autopilot crash: classifies truck as billboard. See Figure 6.
   (b) *Adversarial examples.* Add imperceptible image to panda, NN classifies it as gibbon with 99.9% confidence. See Figure 7.
   Explanation: from http://karpathy.github.io/2015/03/30/breaking-convnets/.

Figure 6: Tesla autopilot crash.

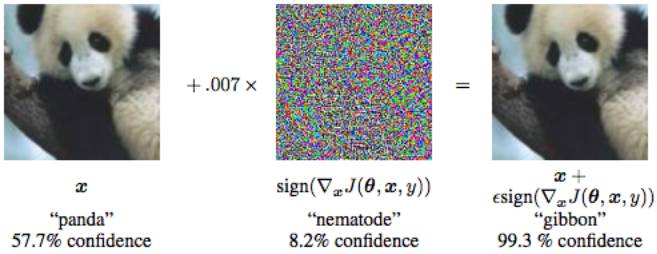


Figure 7: Adversarial example.

Normal ConvNet training: What happens to the score of the correct class when I wiggle this parameter?
Creating fooling images: What happens to the score of (whatever class you want) when I wiggle this pixel?
We compute the gradient just as before with backpropagation, and then we can perform an image update instead of a parameter update, with the end result being that we increase the score of whatever class we want.

(c) *Reproducibility, finicky training.* 99 percent of work is: how to structure architecture correctly, how to tune parameters. Ali Rashimi, NIPS 2017, called it *alchemy*:

> "How many times have you trained deep nets, and felt bad about yourself that they did not work? Wouldn't you like to know how batchnorm reduces internal covariate shift? Wouldn't you like to know what internal covariate shift *is*?""

6. **Automatic feature engineering.**

(a) The classical workflow in applications of supervised machine learning, statistics, data mining, etc., starts by thinking carefully about what features to use.

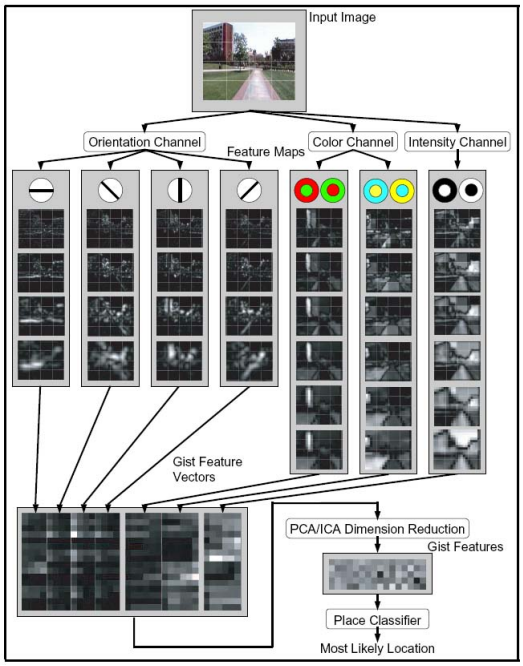


Figure 8: Computer vision features example.

E.g., Computer vision: Gist features. See Figure 8.

(b) This is (1) Time-consuming for humans, (2) Case-by-case, hard to automate, non-adaptive (3) Does not necessarily extract all information.

(c) Deep learning replaces this by learning features automatically. (1) Less time-consuming for humans (even though computationally demanding), (2) Universal workflow, (3) Can always make network huge, and extract all information.

7. **Success criteria.**

(a) There is *only one way* to succeed: reduce classification error.

(b) Even if this costs a lot of computation, and human labor. These are not always reported.

(c) Implications:
"I proved a theorem" → Nobody cares.
"I proved a theorem, *and* it helped me decrease classification error" → Lots of people care.

8. **Workflow: first overfit, then regularize.**

(a) Deep nets are designed to squeeze out every bit of information in the data.

(b) The workflow to do this is: *first overfit, then regularize.*

(c) In order for a class of machine learning models to be successful, it *has to be able to overfit.* Otherwise, it leaves valuable information in the data.
Overfit: Deeper network, wider network, more epochs

(d) Regularize: smaller network, weight decay, dropout, smaller batch size, early stopping, lower learning rate, other hyperparameters

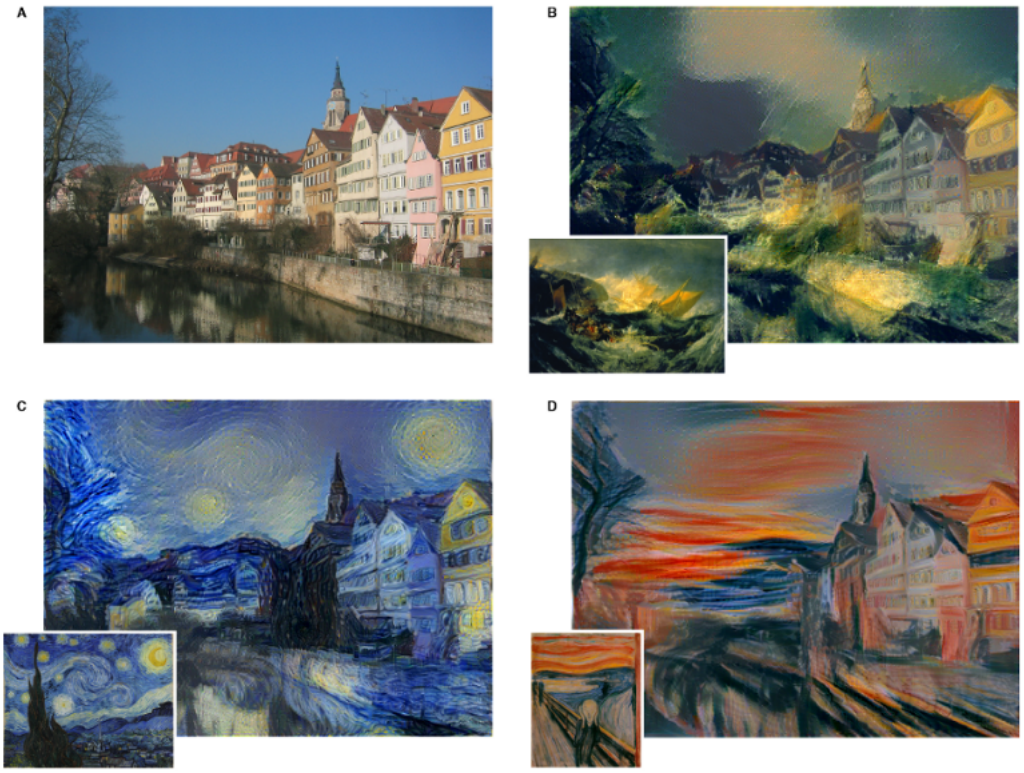


Figure 9: Style transfer (Gatys et al 2015).

1.4 Miscellanea

1. Who cares?
   - Google, Facebook, Microsoft all have AI/Deep learning research groups. Thousands of projects use DL
   - More than 1300 DL startups as of 2017
   - Note: fake startup. Rocket AI.
     How many of the startups have no real basis?
2. Why?
   - Large interest from industry, government, academia. Why?
   - Better-than-human (or just good) performance
     - Computer vision: Classification, Segmentation, Caption generation, Object detection. Image processing: Inpainting, Superresolution, Style transfer (painting: Van Gogh, sketch to photo; design, Figure 9)
     - Speech recognition: Voice commands, Video auto-cap, Real-time translation
     - Text analysis: Translation, Sentiment analysis (predict if review is positive)
     - Robotics: Self-driving cars
     - Fast fluid simulation, fast differential equation solvers.

  - Game playing: Go, Dota
- Potential to automate other tasks?
- Note on better-than-human performance: Computers are already much better than human at many things. e.g. they have much better memory. Do math calculations. Communicate fast. Can be told what to do (and do it), by programming.
- Can do things that were previously thought to be impossible for computers (even though they are trivial for humans)
  Image Classification - humans can do it. But computers can do it faster, more systematically (and more versions).
  And, there are situations where insta-speed is needed.
  This is the first step in scene understanding. When you think about an autonomous agent, it has to figure out where it is.
  It can be combined with other systems. Whenever you have a function that you fit to data, replace it with deep net. e.g., deep RL. any image processing task - deep version.
  Where is the opportunity?

3. Hardware: GPUs and cloud. Camera technology. Internet. Programming languages.
4. Software: Keras. Tensorflow, PyTorch
5. Hype
   - A lot of news coverage. "AI" captures public imagination
   - Many students are interested. "Default topic" for CS research. Typical paper at CS conference "We used DL for X, it worked great"
   - "It can solve any problem".
6. Cautionary notes on the hype
   - Humans have many *cognitive biases* (see Tversky & Kahneman, 2011 Nobel Prize in Economics)
   - *Availability bias:* DL makes choosing research topic easy
   - *Affect bias:* "deep learning" vs "neural networks"
   - *Overconfidence:* "AI revolution"
   - *Neutral reference point:* take amazing things for granted
   - A lot of tuning in the background, dishonest presentation. "AI learns preiodic table in 8 hours"
     "The 0.1% mentality." At a big company, only need to improve tiny bit to keep job. Tuning is life.
7. Given the large hardware, power, and human *costs* of the using neural networks it would be extremely valuable to have some guidance about when they are expected to work. Many of the other algorithm that are deployed in applications are known to have a good performance under some conditions. The value of these heuristics and results is that one can prioritize the algorithms and save resources.

   Here are a few widely used algorithms and statistical methods and the settings under which they are known to work:

   (a) *PageRank.* The core ranking algorithm used by Google's search engine, this is a linear algebra based method. The computational convergence of the algorithm in guaranteed based on strong results from numerical linear algebra.

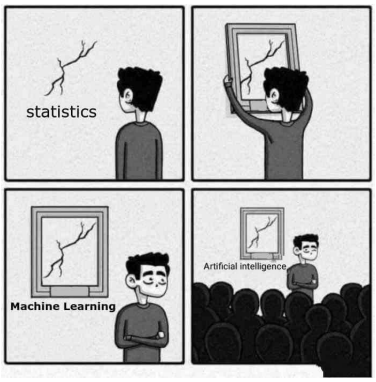


Figure 10: Statistics, ML, and AI. A humorous perspective.

(b) *Linear regression.* This is a classical method for prediction and statistical inference. The computation is based on simple linear algebra. The accuracy of the algorithm for estimation is known in linear models.

In contrast there is much less known about the settings in which deep neural networks will work. We discuss some more specific questions below.

8. **Optimization.**
   Simplest method (SGD) basically "works". How? Why is it so easy to optimize? Can it be made better?
9. **Generalization.**
   Why not overfit? But: see concerns about overfitting to benchmark datasets. How can we reduce overfitting even more?
   Empirical observations in the paper by Zhang et al, "Understanding DL requires rethinking generalization", ICLR 2017
   (a) NNs fit everything. [This was not 100% reproducible by other groups later.] Classical learning theory does not apply.
   (b) SGD is regularization
   Other regularization doesn't help much
   (c) NNs are not stable
   (d) Optimization is empirically easy even if the resulting model does not generalize.
10. Comparing and contrasting deep learning and statistics. See also Figure 10
    DL and stat both fit models, predict on data. However, the problems they address are very different: see Table 1.
11. Limitations:
    People don't learn like that
    There are many documented examples (and many more undocumented ones), where simpler methods work better.
    Dexterity and manipulation
    They are not intelligent!
12. Thought leaders in the tech world all bet on AI:
    Bill Gates:

Table 1: DL and statistics

	Deep learning	Statistics
Sample size	Huge	Not too large
Structure of Data	High	Varies
Noise	None	Can be high
Difficulty for human	Easy	Can be hard
Computational Need	Massive	Varies
Parameters	As many as data	Few
Interpretability	Low	High
Guarantees	None	Under assumptions
Human tuning	Effortful	Easy
Overall Cost	High	Lower

What technology are you most looking forward to in the next 10 years and what impact do you think it could have?
Bill Gates: The most amazing thing will be when computers can read and understand the text like humans do. Today computers can do simple things like search for specific words but concepts like vacation or career or family are not "understood". Microsoft and others are working on this to create a helpful assistant. It has always been kind of a holy grail of software particularly now that vision and speech are largely solved. Another frontier is robotics where the human ability to move and manipulate is amazing and experts disagree on whether it will take just a decade or a lot longer (Brooks) to achieve the equivalent.

Jeff Bezos:

Embrace External Trends
The outside world can push you into Day 2 if you wont or cant embrace powerful trends quickly. If you fight them, youre probably fighting the future. Embrace them and you have a tailwind. These big trends are not that hard to spot (they get talked and written about a lot), but they can be strangely hard for large organizations to embrace. Were in the middle of an obvious one right now: machine learning and artificial intelligence. Over the past decades computers have broadly automated tasks that programmers could describe with clear rules and algorithms. Modern machine learning techniques now allow us to do the same for tasks where describing the precise rules is much harder.
Inside AWS, were excited to lower the costs and barriers to machine learning and AI so organizations of all sizes can take advantage of these advanced techniques. Watch this space. Much more to come.

13. Intrepretability: excellent reference https://distill.pub/2018/building-blocks/
14. Creative architecture design, Figure 11

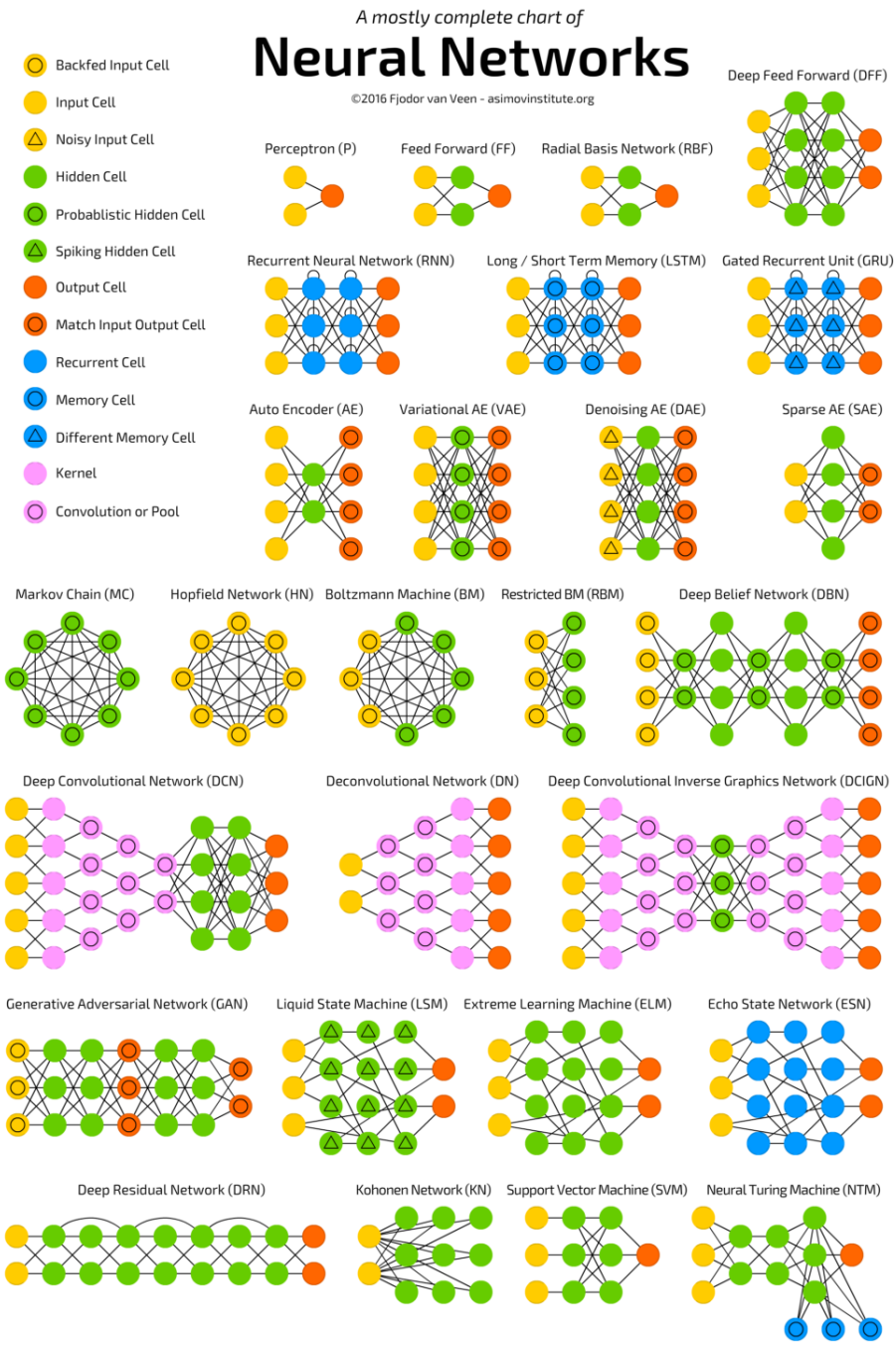


Figure 11: Creative architectures.

2 Convolutional neural networks (CNNs)

2.1 The problem and model

1. Real data has structure. Nets can exploit this. e.g., images have a lot of structure: always a small number of objects, composed of smaller objects, strong correlations between pixels.
   Computer vision
   (a) *Image classification*: e.g., recognize faces
   (b) *Object detection*: e.g., find cars in an environment
   (c) *Style transfer*: e.g., design new objects based on existing styles (Figure 9)

   Image dimensions are very large, so fully connected networks have too many parameters. *Convolutional neural networks (CNNs)* are used instead. They exploit the "structure" of natural images in a more efficient way.
2. *Image* X: $n \times n$. e.g., an image of a road scene with cars.
3. *Convolutional Layer:* The structure of a convolutional layer is as follows
   (a) *Filter* F: $f \times f$ image, of small size, 3×3, 5×5, etc. e.g., an edge detector.
   The *convolution* operation is denoted as
   $$X * F$$
   This takes all possible inner products of F with *patches* of X. It creates an $(n - f + 1) \times (n - f + 1)$ image.
   Formally, an entry i, j of $X * F$ is defined as
   $$X * F[i, j] = \mathrm{tr}\left\{X[i : i + f - 1, j : j + f - 1] \cdot F\right\}$$
   (where $A \cdot B$ is the usual matrix multiplication)
   Easy to extend to "*tensors*", where you have several *channels* of an image (e.g., color channels). A filter has the same depth as the number of channels of the previous layer, and the inner products are computed and summed over all channels.
   (b) *Padding*:
   Pad outside of X with zeros to keep the dimension of images fixed.
   After padding with p zeros, the output is $(n + 2p - f + 1) \times (n + 2p - f + 1)$
   Filter size is usually chosen as an odd integer, so we can pad with $p = (f - 1)/2$ pixels everywhere, to keep the dimension of images fixed.
   (c) *Strided convolutions*:
   Jump over some steps, so restrict to $X * F[i, j]$ to $i, j = 1, 1 + s, 1 + 2s, \ldots$. Then the output will have size $\lfloor \frac{n+2p-f+1}{s} \rfloor \times \lfloor \frac{n+2p-f+1}{s} \rfloor$
   (d) Usually take multiple filters. These become the channels in the next layer. With an image of size $n \times n \times L$, and k layers of size $f \times f \times L$, get an output tensor of size $(n - f + 1) \times (n - f + 1) \times k$
   Number of parameters is much smaller than for fully connected networks. "Parameter sharing", sparse connections.
   Why convolutions? Vaguely related to local translation invariant structure of images, and *symmetry*. More precisely?
   (e) Then, add bias and apply ReLU.
   In deep networks, in the first layers typically we keep an equal height and width, and add more channels. Then reduce H/W, by increasing stride.

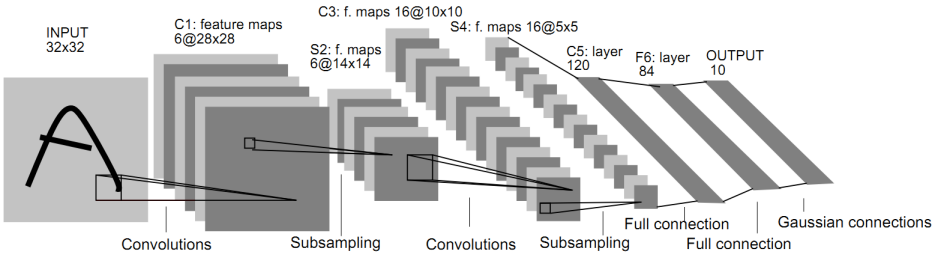


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Figure 12: LeNet-5. (LeCun, 1998)

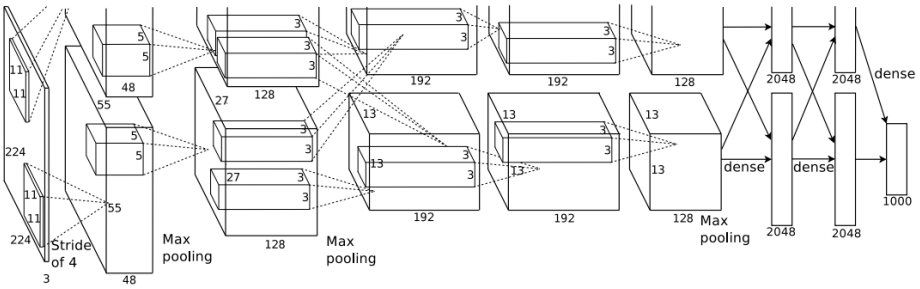


Figure 13: AlexNet. Krizhevsky et al (2012, NIPS), Won ILSVRC 2012.

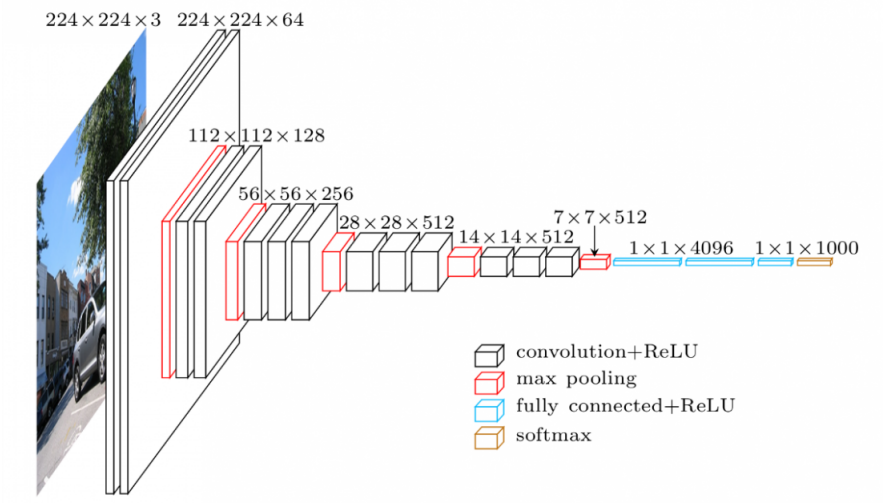


Figure 14: VGG. (Simonyan, Zisserman, 2015).

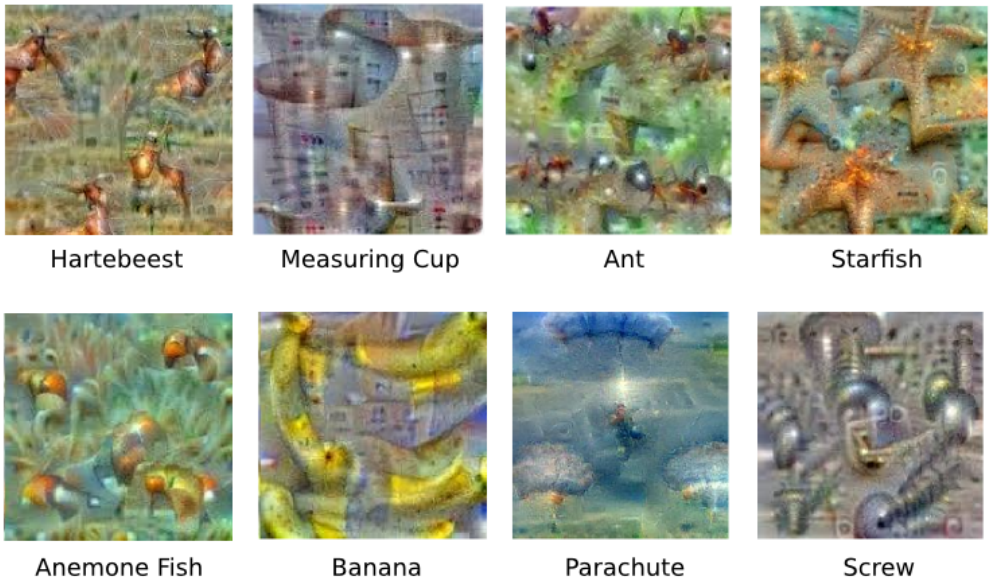


Figure 15: Images that maximize activation. The neurons selected for these images are the output neurons that a DNN uses to classify images as bananas etc.

4. *Pooling.* Reduces dimension of an image by "pooling" across neighboring activations.
   Max-pooling with parameters f, p

$$X_P[i,j] = \max_{i \le k \le i+f-1, j \le l \le j+f-1} X[k,l]$$

   And restrict to $i, j = 1, 1+s, 1+2s, \ldots$.
   Average-pooling used to be popular, but the non-linear max pooling seems to work better recently.
5. The last few layers are typically fully connected.
   Simple example: "*LeNet*" type networks, starting from 1980s (Figure 12).
   More recent examples: AlexNet (Figure 13). VGG (Figure 14).
6. Visualize NNs, by finding images that maximize activation: hallucinogenic images (Figure 15).
7. Excellent reference on feature visualization: https://distill.pub/2017/feature-visualization/
   Live demo of digit visualization: http://scs.ryerson.ca/~aharley/vis/conv/
8. See also http://colah.github.io/posts/2014-07-Conv-Nets-Modular/,
   http://colah.github.io/posts/2014-07-Understanding-Convolutions/
   Yosinski's Deepvis toolbox: http://yosinski.com/deepvis

2.2 Other aspects

1. R-CNN: Regions with CNN features (Girshick et al, 2013, Rich feature hierarchies for accurate object detection and semantic segmentation). Figure 16.

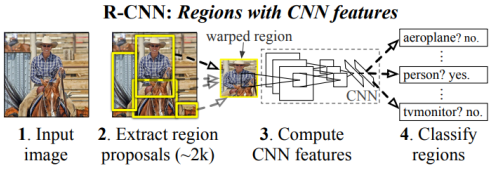


Figure 1: Object detection system overview. Our system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large convolutional neural network (CNN), and then (4) classifies each region using class-specific linear SVMs. R-CNN achieves a mean average precision (mAP) of **53.7% on PASCAL VOC 2010**. For comparison, [39] reports 35.1% mAP using the same region proposals, but with a spatial pyramid and bag-of-visual-words approach. The popular deformable part models perform at 33.4%. On the 200-class **ILSVRC2013 detection dataset, R-CNN's mAP is 31.4%**, a large improvement over OverFeat [34], which had the previous best result at 24.3%.

Figure 16: R-CNN.

Relies on "Selective Search for Object Recognition", Uijlings, van de Sande (2013) for region proposal.

Once the proposals are created, R-CNN warps the region to a standard square size and passes it through to a modified version of AlexNet (the winning submission to ImageNet 2012 that inspired R-CNN), as shown above.

On the final layer of the CNN, R-CNN adds a Support Vector Machine (SVM) that simply classifies whether this is an object, and if so what object. This is step 4 in the image above.

Later: Fast R-CNN, Mask R-CNN. See

https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34e

2.3 Training

1. In principle, training proceeds in the same way via backpropagation
2. *Transfer learning* is an important way to reduce computational cost. You can use the first layers a pre-trained standard architecture (such as VGG-19) as the starting point of your architecture. You can extend it to suit your problem by adding new layers.

   It is often observed that the features learned on one task, such as ImageNet classification, can be quite transferable to other problems, such as image segmentation.
3. *Data Augmentation.* Done in a streaming fashion.

   Mirroring, Shifting, Random cropping. Also: *Rotation, Shearing*

   Color shifting. From the AlexNet paper:

   > Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components, with magnitudes proportional to the corresponding eigenvalues times a random variable drawn from a Gaussian with mean zero and standard deviation 0.1.

This scheme approximately captures an important property of natural images, namely, that object identity is invariant to changes in the intensity and color of the illumination.

2.4 Graph CNN

- Convolutional Neural Networks are efficient architectures in image and audio recognition tasks, thanks to their ability to exploit the local translational invariance of signal classes over their domain. How about signals defined on more general domains?
- 3.1 Graph Signal Processing.

  "The emerging field of GSP aims at bridging the gap between signal processing and spectral graph theory, a blend between graph theory and harmonic analysis. A goal is to generalize fundamental analysis operations for signals from regular grids to irregular structures embodied by graphs. Standard operations on grids such as convolution, translation, filtering, dilatation, modulation or downsampling do not extend directly to graphs and thus require new mathematical definitions while keeping the original intuitive concepts."

  D. Shuman, S. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The Emerging Field of Signal Processing on Graphs: Extending High-Dimensional Data Analysis to Networks and other Irregular Domains. 2013.
- From Bruna, Zaremba, Szlam, LeCun "Spectral Networks and Deep Locally Connected Networks on Graphs" (2014)

  On a regular grid, a CNN is able to exploit several structures that play nicely together to greatly reduce the number of parameters in the system:

  – The translation structure, allowing the use of filters instead of generic linear maps and hence weight sharing.
  – The metric on the grid, allowing compactly supported filters, whose support is typically much smaller than the size of the input signals.
  – The multiscale dyadic clustering of the grid, allowing subsampling, implemented through stride convolutions and pooling.

  In many contexts, however, one may be faced with data defined over coordinates which lack some, or all, of the above geometrical properties. For instance, data defined on 3-D meshes, such as surface tension or temperature, measurements from a network of meteorological stations, or data coming from social networks or collaborative filtering, are all examples of structured inputs on which one cannot apply standard convolutional networks. Another relevant example is the intermediate representation arising from deep neural networks. Although the spatial convolutional structure can be exploited at several layers, typical CNN architectures do not assume any geometry in the feature dimension, resulting in 4-D tensors which are only convolutional along their spatial coordinates.
- Spatial Construction
  – Local, multiscale, hierarchical,
  – Graph $G = (\Omega, W)$. Vertices Ω are a set of size m, and edge weights are collected in an $m \times m$ nonnegative matrix W
  – Local: Neighborhood $N_\delta(j) = \{i : W_{ij} > \delta\}$. Use only filters restricted to neighborhoods.
  – Multiscale: "In this work we will use a naive agglomerative method."
  – Hierarchical: Exactly what you think

- Spectral Construction
  - Combinatorial Laplacian: $L = D - W$, where $D = \text{diag}(W1)$ is the diagonal matrix of degrees.
    Graph Laplacian $\mathcal{L} = I - D^{-1/2}WD^{-1/2}$.
  - Smoothness functional for signal x on graph:

    $$\|\nabla x\|_W^2(i) = \sum_j W_{ij}[x(i) - x(j)]^2$$

    So

    $$\|\nabla x\|_W^2 = \sum_{i,j} W_{ij}[x(i) - x(j)]^2 = 2x^\top Lx$$

  - Global min: $v_0 = m^{-1/2}1$.
    Successive minima:

    $$v_i = \arg \min_{x \in \mathbb{R}^m, x \perp v_j, j<i} \frac{\|\nabla x\|_W^2}{\|x\|^2}$$

    Eigenvectors of L with eigenvalues λ_i "allow the smoothness of a vector to be read off from the eigenvalues, equivalently as the Fourier coefficients of a signal defined in a grid. Thus, just an in the case of the grid, where the eigenvectors of the Laplacian are the Fourier vectors, diagonal operators on the spectrum of the Laplacian modulate the smoothness of their operands."
  - "3.2 Extending Convolutions via the Laplacian Spectrum"
    Given a weighted graph, we can try to generalize a convolutional net by operating on the spectrum of the weights, given by the eigenvectors of its graph Laplacian.
    - Start with K layers, transforming input vectors x_k of size $m \times f_{k-1}$ to output x_{k+1} of size $m \times f_k$ (no spatial subsampling)

      $$x_{k+1,j} = \sigma(V \sum_{i=1}^{f_k} F_{kij} V^\top x_{ki})$$

      where V is the matrix of eigenvectors of L, and F_{kij} is a diagonal matrix.
    - Issues: most graphs have meaningful eigenvectors only for the very top of the spectrum
  - "3.3 Rediscovering standard CNNs". A simple, and in some sense universal, choice of weight matrix in this construction is the covariance of the data. It turns out that with this weights, gNN approximately recovers CNN.
  - Spatial locality: Smoothness in the spectral domain. "A particularly simple and naive choice consists in choosing a 1-dimensional arrangement, obtained by ordering the eigenvectors according to their eigenvalues. In this setting, the diagonal of each filter F_{kij} (of size at most m) is parametrized by

    $$diag(F_{kij}) = K\alpha_{kij}$$

    where K is a $d \times q_k$ fixed cubic spline kernel and α_{kij} are the spline coefficients.
- Improvements: "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering" Michal Defferrard, Xavier Bresson, Pierre Vandergheynst. (Figure 17)

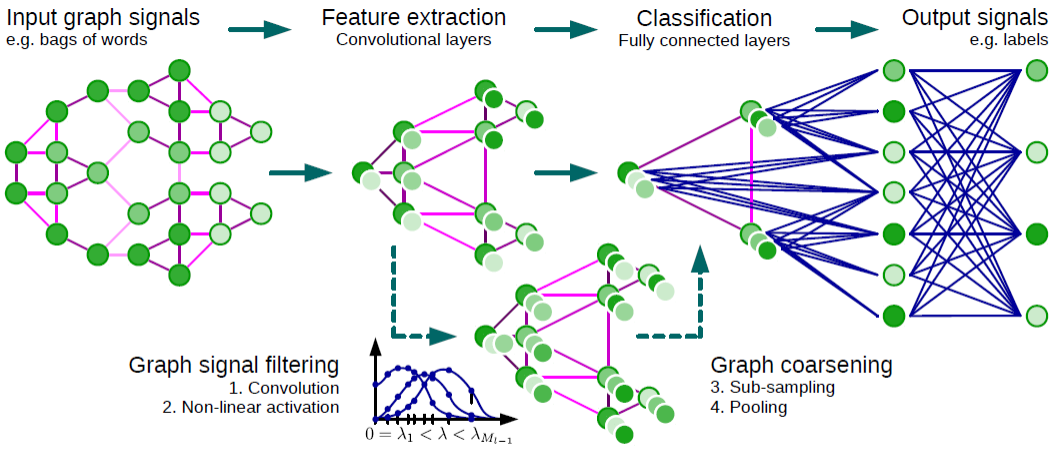


Figure 1: Architecture of a CNN on graphs and the four ingredients of a (graph) convolutional layer.

Figure 17: Graph CNN.

1. Limitations: "the dominant cost is the need to multiply the data by U twice (forward and inverse Fourier transforms)". "no precise control over the local support of their kernels"
2. 2 Proposed Technique
   Generalizing CNNs to graphs requires three fundamental steps: (i) the design of localized convolutional filters on graphs, (ii) a graph coarsening procedure that groups together similar vertices and (iii) a graph pooling operation that trades spatial resolution for higher filter resolution.
3. They parametrize the filters F as polynomials of the eigenvalues, $g_\theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k \Lambda^k$, where Λ is the diagonal matrix of eigenvalues of the laplacian.
   Spectral filters represented by K-th order polynomials of the Laplacian are exactly K-localized.
4. Moreover, to compute it efficiently, use a recurrence like Chebyshev polynomials.
5. Training. Todo.

$$\begin{aligned} y &= g_\theta(L)x \\ &= \sum_{k=0}^{K-1} \theta_k T_k(\tilde{L})x \\ &= [\bar{x}_0, \dots, \bar{x}_{K-1}]\theta \end{aligned}$$

The j-th output feature map the sample s is given by

$$y_j = \sum_{i=1}^{F_{in}} g_{\theta_{ij}}(L)x_{si}$$

where the x_{si} are the input feature maps and the $F_{in} \times F_{out}$ vectors of Chebyshev coefficients $\theta_{ij} \in \mathbb{R}^K$ are the layers trainable parameters. When training multiple convolutional layers with the backpropagation algorithm, one needs the two gradients

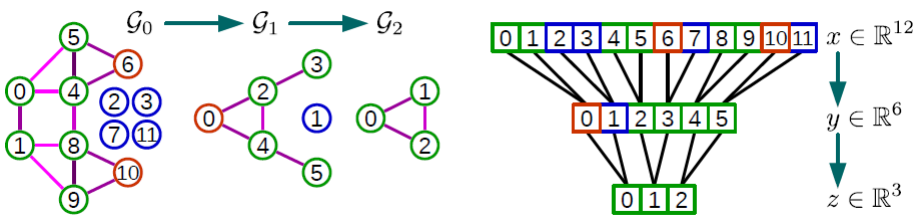


Figure 18: Graph Coarsening and Pooling.

$$\partial E/\partial \theta_{ij} = \sum_{s=1}^{S} [\bar{x}_{s,i,0}, \ldots, \bar{x}_{s,i,K-1}]^T \partial E/\partial \theta_{ij}$$

and

$$\partial E/\partial x_{si} = \sum_{j=1}^{F_{out}} g_{\theta_{ij}}(L) \partial E/\partial y_{sj}$$

where E is the loss energy over a mini-batch of S samples. Each of the above three computations boils down to K sparse matrix-vector multiplications and one dense matrix-vector multiplication for a cost of $O(K|E|F_{in}F_{out}S)$ operations. These can be efficiently computed on parallel architectures by leveraging tensor operations.

6. 2.2 Graph Coarsening. (Figure 18)
   Graclus [9], built on Metis [16], uses a greedy algorithm to compute successive coarser versions of a given graph and is able to minimize several popular spectral clustering objectives, from which we chose the normalized cut [30].
   Graclus' greedy rule consists, at each coarsening level, in picking an unmarked vertex i and matching it with one of its unmarked neighbors j that maximizes the local normalized cut $W_{ij}(1/d_i + 1/d_j)$. The two matched vertices are then marked and the coarsened weights are set as the sum of their weights. The matching is repeated until all nodes have been explored. [Form of hierarchical clustering.]
7. 2.3 Fast Pooling of Graph Signals
   Pooling operations are carried out many times and must be efficient. After coarsening, the vertices of the input graph and its coarsened versions are not arranged in any meaningful way. Hence, a direct application of the pooling operation would need a table to store all matched vertices. That would result in a memory inefficient, slow, and hardly parallelizable implementation. It is however possible to arrange the vertices such that a graph pooling operation becomes as efficient as a 1D pooling. We proceed in two steps: (i) create a balanced binary tree and (ii) rearrange the vertices.

- Convolutional Neural Network Architectures for Signals Supported on Graphs, Gama, Marques, Leus, Ribeiro. 2018.
  SELECTION GRAPH NEURAL NETWORKS
  Graph G with nodes n
  Features x_0^f, (or just x, with entries x_n at node n)
  Filters h_1^{fg} (vectors of arbitrary dimension K)
  Shift operator S, which is a square matrix having the same sparsity pattern of the graph; i.e., we can have $S_{mn} \neq 0$ if and only if $(n, m) \in E$ or $m = n$. The shift operator

is a stand in for one of the matrix representations of the graph. Commonly used shift operators include the adjacency matrix A with nonzero elements $A_{mn} = W(n, m)$ for all $(n, m) \in E$, the Laplacian $L := diag(A1) - A$ and their normalized counterparts.

Intermediate features u_1^{fg} with components

$$[u_1^{fg}] := [h_1^{fg} *_S x_0^g]_n := \sum_{k=0}^{K_1-1} [h_1^{fg}]_k [S^k x_0^f]_n$$

where we have used $*_S$ to denote the graph convolution operation on S.

The graph filter is a generalization of the Chebyshev filter in Deferrard et al. More precisely, if G is an undirected graph, and we adopt the normalized Laplacian as the graph shift operator S, then it boils down to a Chebyshev filter.

A. Selection Sampling on Graph Convolutional Features

The challenge in generalizing CNNs to GNNs arises beyond the first layer. After implementing the sampling operation, the signal is of lower dimensionality and can no longer be interpreted as a signal supported on S.

Resolving mismatched supports is a well-known problem in signal processing whose simplest and most widely- used solution is zero padding.

Sampling is an operation that selects components of a signal. To explain the construction of convolutional features on graphs, it is more convenient to think of sampling as the selection of nodes of a graph which we call active nodes.

B. Selection Sampling and Pooling

The pooling stage requires that we redefine the summary and sampling operations in (4) and (5). Generalizing the summary operation requires redefining the aggregation neighborhood.

Thus, at layer l we introduce an integer a_l to specify the reach of the summary operator and define the a_l-hop neighborhood of n as

$$n_l = [m : [S_l^{(k)}]_{nm} \neq 0, \text{for some } k \leq a_l]$$

This number of components is reduced at the pooling stage in which the values of a group of neighboring elements are aggregated to a single scalar using a possibly nonlinear summarization function.

To complete the pooling stage we follow this with a down- sampling operation.

- Why graph convolutions? Permutation invariance. Ruiz, Gama, Marques, Ribeiro, Median activation functions for graph neural networks, in ICASSP 2019. [?]

  Graph filters exploit the inherent symmetries of a graph. If different parts of a graph look the same a graph filter will process the signal in the same manner because the graph can be permuted onto itself.

  [First need to understand scattering transform.]

- Why nonlinear GNNs, and not linear? stability to graph deformations. Gama, Ribeiro, Bruna, Diffusion scattering transforms on graphs, ICLR 2019.

  This is still true if different parts of the graph look similar without necessarily being identical. This stability is present in GNNs but it is not present in graph filters.

2.5 Shapes beyond images, invariance

- Classical CNNs are designed for images. However, in computer vision we often have other types of shapes. For instance, we may have to deal with 3D shapes (when interacting with an environment), or we may have spherical data (e.g., a self-driving car takes pictures on a sphere)
- To design NN architectures for such shapes, we need to replace convolutions with the appropriate generalization
- We can rely on groups. We are thinking of objects x that are acted upon by members g of a group G. For instance, images are acted upon by 2D translations. We want to extract filters (features) $f(x)$ from the group that are equivariant to the action of the group. Specifically, we want to ensure that there is a group G' such that for every g, there is a $g' \in G'$ for which

$$f(gx) = g' f(x)$$

- One approach to construct this is group convolutions. There we define

$$(f *_G h)(x) = \int_{g \in G} f(ge) h(g^{-1}x) dg$$

for some fixed element $e \in G$.

The familiar convolution in 2D is a special case, where $G = (\mathbb{R}, +)$, and $e = 0$, so that

$$(f *_G h)(x) = \int_{g \in \mathbb{R}^2} f(g) h(x - g) dg$$

One can check that this leads to equivariant maps.

- The next step is to design architectures that implement group convolutions for various specific groups.

  An important example is the group 3D rotations, $SO(3)$, acting on the sphere. See work by Cohen and Welling (ICLR 2018), and Esteves, Allen-Blanchette, Makadia, Daniilidis [EAMD] (ECCV 2018)

  A key point is that discretization is hard, because a sampling of the sphere with well distributed and compact cells with transitivity does not exist.
- Instead, EAMD work with spherical signals using the spherical harmonics (Fourier transform on the sphere). Then convolution is evaluated in the spectral domain, by multiplication.

  There are many other details. They design architectures that are smaller and do not need data augmentation. They apply them to retrieval and classification of 3D models (retrieval = find similar objects)
- This has many applications. In later work, Esteves, Sud, Luo, Daniilidis, Makadia, use it to design 3D equivariant image embeddings.

  The basic problem is that 2d image embeddings learned by standard CNN are not equivariant to the 3d rotations of the underlying 3d objects of which we are taking the images. However, using spherical CNN one can construct approximately equivariant embeddings.

  The idea is that we try to invert the spherical CNN. Let $s(x)$ be a spherical CNN that is equivariant to rotation of the 3d object x, that is $s(rx) = rs(x)$. Let $c(x)$ be the 2d projection of x.

If we learn a feature map f that inverts the spherical CNN, in the sense that $f(c(x)) = s(x)$, then it is easy to check that f will be equivariant.

This can be accomplished by designing an appropriate invariant loss function $L(x, y)$, and training a neural network for f

- Another application is to design efficient algorithms for processing multi-view data. Instead of a single layer of pooling, we can construct views that are on a subgroup of SO(3), and use a group convolution approach. See Esteves, Xu, Allen-Blanchette, Daniilidis (2019)
- An older approach to equivariance with respect to continuous Lie groups is using generators. Equivariance in the context of pattern recognition was first introduced by Amari in 1978 using the Lie generators of the underlying Lie group action.

  Below: [Nordberg, Granlun, 1996]

  A generator L for a Lie group is a linear transform such that the action of the group on a vector v can be written as $\exp(xL)v$, where $\exp(M) = \sum_{k=0}^{\infty} L^k/k!$ is the operator exponential.

  This is just for one-parameter groups, but for multiparameter groups we can have similarly $\exp(x_1L_1 + \ldots + x_kL_k)v$.

  "Lie theory [I], [6], shows that there is a one-to-one correspondence between a continuous operator group M and its generator L (scale factors disregarded). More complex operator groups can be defined using multiple, linearly independent, generators. For example, the group of 3D rotations has three generators."

  "Given the two groups M and N, represented by their corresponding generators, we would like to character- ize all functions f that are equivariant with respect to the groups. Nordberg [5] proves that this can be done assuming that f has a Taylor expansion into a power series of v The proof gives a sufficient condition ex- pressed in terms of a linear equation in the coefficients of each term in the Taylor expansion."

  Specifically, want f such that for any x, there is y such that

  $$f(\exp(xL_M)v) = \exp(yL_N)f(v)$$

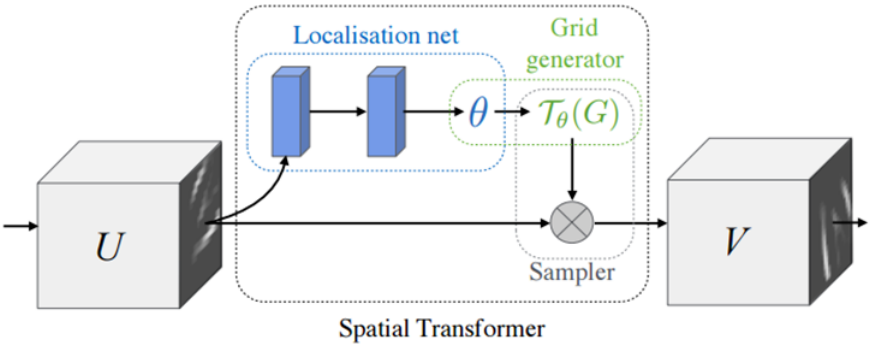
  [Requires continuous group, construct generator, solve equations to find explicit equivariant transform]

  "Lie group transformations could be converted to translations in canonical coordinates if the corresponding Lie generators were linearly independent. This is true, for example, for rotation and scalings but not for translations and rotations. Moreover, this constrains the maximum dimension of the subgroup to be the dimension of the space where the group is acting (two in the case of images)"
- "Closely related to the (in/equi)variance work is work in steerability, the interpolation of responses to any group action using the response of a finite filter basis. An exact steerability framework was proposed in [9] (The design and use of steerable filters, 1991), where rotational steerability for Gaussian derivatives was explicitly computed. A method of approximating steerability by learning a lower dimensional representation of the image deformation from the transformation orbit and the SVD was proposed by [10]."

  Short description of the idea: Gaussian $G(x, y) = \exp(-(x^2 + y^2))$. Rotation operator $(...)^\theta$, st for any function $f(x, y)$, $f(x, y)^\theta$ is f rotated through an angle θ about the origin.

  [G can be viewed as a bump detector]

  The first x derivative of a Gaussian G_1^0 is



A Spatial Tranfomer module

Figure 19: Spatial Transformer Networks.

$$G_1^0 = \partial G/\partial x = -2xG$$

The same function, rotated 90, is

$$G_1^{90} = \partial G/\partial y = -2yG$$

[These functions can be viewed as edge detectors H/V]

Now

$$G_1^\theta = \cos(\theta)G_1^0 + \sin(\theta)G_1^{90}$$

Since G_1^0, G_1^{90} span the set of G_1^θ filters, we call them basis filters. The cos,sin are interpolation functions.

Because convolution is linear, we can sythesize images filtered at arbitrary orientations by taking linear combinations of the images filtered with the basis.

Let A be image. $R_1^\theta = G_1^\theta * A$. Then

$$R_1^\theta = \cos(\theta)R_1^0 + \sin(\theta)R_1^{90}$$

So we can compute convolutions with rotated filters easily.

[Only works for constructed, not learned filters.]

- To read: proposal, polar transformer networks

2.6 Spatial Transformer Networks

1. Spatial Transformer Networks (2015)

   Transform and simplify image before feeding it into CNN

   Correct for pose normalization (scenarios where the object is tilted or scaled) and spatial attention (bringing attention to the correct object in a crowded image)

   See Figure 19

Components:

A localization network. Input: volume, Output: parameters of the spatial transformation that should be applied. The parameters can be 6 dimensional for an affine transformation.

The creation of a sampling grid that is the result of warping the regular grid with the affine transformation created in the localization network.

A sampler whose purpose is to perform a warping of the input feature map.

See description at adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

3 Recurrent neural networks (RNNs)

3.1 Setup

1. *Recurrent neural networks (RNNs)* are a class of models appropriate for sequential data. e.g.,

   text, speech, video, DNA sequence,

   They exploit that the relation between temporal units is often invariant in time. Similar purpose as time series models in statistics.

2. *Input sequence* x, e.g., sentence:

   $$x = x^{<1>}, x^{<2>}, \dots, x^{<t>}, \dots, x^{<T_x>}$$

   Has *elements* $x^{<i>}$, e.g., words

   (a) x - "The cat is white"

   (b) $x^{<1>}$ - "The"; $x^{<2>}$ - "cat"; $x^{<3>}$ - "is"; $x^{<4>}$ - "white"

   Output sequence, e.g., word labels:

   $$y = y^{<1>}, y^{<2>}, \dots, y^{<t>}, \dots, y^{<T_y>}$$

   (a) $y^{<1>}$ - pronoun; $y^{<2>}$ - noun; $y^{<3>}$ - verb; $y^{<4>}$ - adjective

   (Do not need to have the same length)

3. *Word representations*: use *one-hot vectors* to represent words. Each element (word) will be a vector

   $$e_j = (0, \dots, 0, 1, 0, \dots, 0).$$

   The length of the vector is the size of the *dictionary*.

   In principle, this reduces the problem to a supervised learning problem. However, the direct application of NNs is inefficient. Does not use sequential structure.

4. RNN works sequentially:

   (a) Start with activations $a^{<0>}$, say

   $$a^{<0>} = 0$$

   (b) At every step: construct activations $a^{<t>}$
   Use previous activations $a^{<t-1>}$, and current input element $x^{<t>}$, to predict output element $y^{<t>}$.
   This prediction will be governed by parameters W_a, shared across time.

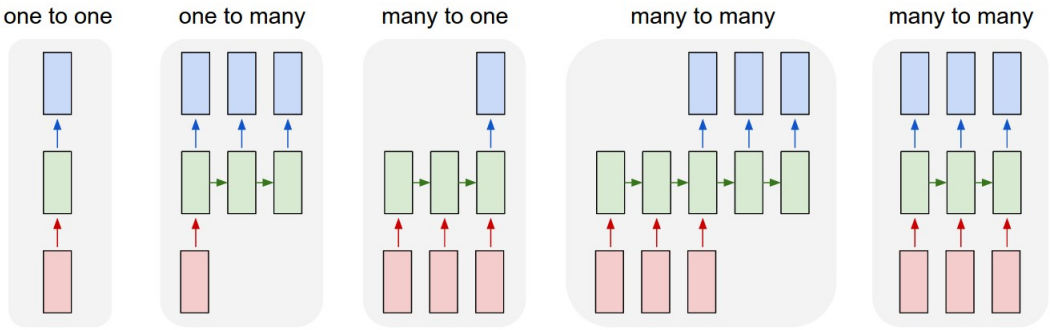


Figure 20: RNNs.

$$a^{<t>} = \sigma_1(W_a[a^{<t-1>}, x^{<t>}] + b_a)$$
$$\hat{y}^{<t>} = \sigma_2(W_y a^{<t>} + b_y)$$

As described by Andrej Karpathy "RNNs combine the input vector x with their state vector a with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables. Viewed this way, RNNs essentially describe programs."

5. Training: Backprop proceeds as before.
   Loss is often defined elementwise: $L(\hat{y}, y) = \sum_{t=1}^{T} \ell(\hat{y}^{<t>}, y^{<t>})$
6. There are many variants: We saw *Many-to-many* with equal input and output lengths.
   One-to-Many: e.g., start with one input element, and produce entire sequence. eg., music generation. Feed each output element forward too $\hat{y}^{<t>}$
   See also Figure 20
7. *Gated Recurrent Unit (GRU)*: Can help better capture long-term dependence (Cho et al, 2014, Chung et al, 2014)
   Add *memory cell* c, with elements $c^{<t>}$. This remembers the state.
   e.g., which person is the current part of the sentence about?
   Also add *candidate update cell* $\tilde{c}^{<t>}$. This is the potential updated state.

$$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$$

   e.g., if the previous state is a name, and new word is a name, then candidate update is the new name
   Gate, or *update gate*: determines whether or not there is an update (think 0, 1)

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

   e.g., if previous state is a name, under what conditions should I switch topics?
   (a) John met Jane who was two years older.
   (b) John met Jane but did not talk much.

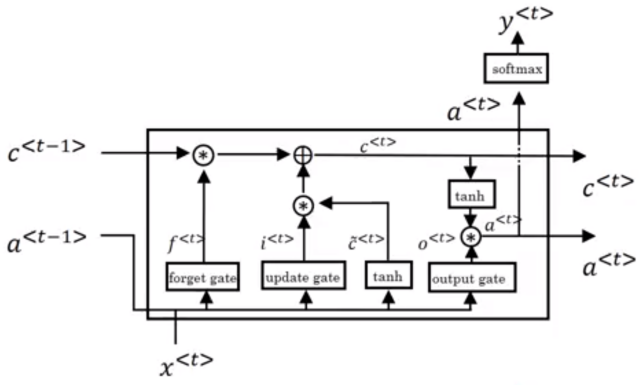


Figure 21: LSTM. Courtesy of Andrew Ng's course.

Update rule:

$$c^{<t>} = \Gamma_u \odot \tilde{c}^{<t>} + (1 - \Gamma_u) \odot c^{<t-1>}$$

Gate = 0, don't update. Gate = 1, update.

The entire GRU is a bit more complicated. Also includes *relevance gate* Γ_r, tells us how relevant is $\tilde{c}$ to computing the next c

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[\Gamma_r \odot c^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u &= \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \\
\Gamma_r &= \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \\
c^{<t>} &= \Gamma_u \odot \tilde{c}^{<t>} + (1 - \Gamma_u) \odot c^{<t-1>}
\end{aligned}$$

8. *Long Short Term Memory (LSTM)* (Hochreiter, Schmidhuber, 1997):
   Similar, but more powerful (and more complicated). There is a separate *forget gate* Γ_f, to control the update rule. There is a separate *output gate* Γ_o, to control output.

$$\begin{aligned}
\tilde{c}^{<t>} &= \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \\
\Gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\
\Gamma_f &= \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \\
\Gamma_o &= \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \\
c^{<t>} &= \Gamma_u \odot \tilde{c}^{<t>} + \Gamma_f \odot c^{<t-1>} \\
a^{<t>} &= \Gamma_o \odot \tanh(c^{<t>})
\end{aligned}$$

   See Figure 21.

9. Karpathy: "An important point to realize is that even if your inputs/outputs are fixed vectors, it is still possible to use this powerful formalism to process them in a sequential manner. For instance, [we can] learn a recurrent network policy that steers its attention around an image; In particular, learn to read out house numbers from left to right (Ba et al. 2014, Multiple Object Recognition with Visual Attention)."

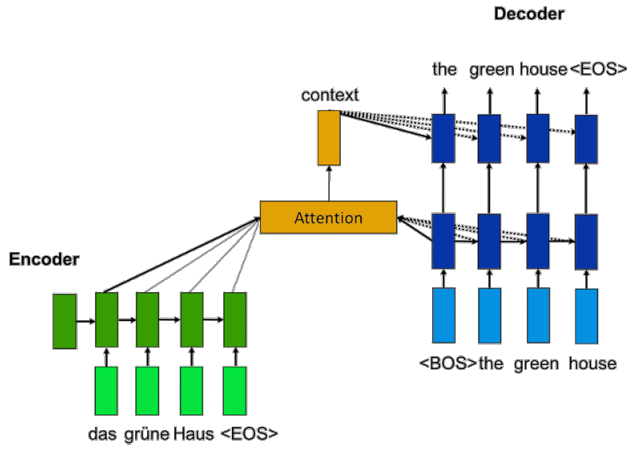


Figure 22: Attention Model.

10. Attention model. Bahdanau et al, Neural Machine Translation by Jointly Learning to Align and Translate (2014): Figure 22

    See https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html for a detailed description

11. Some applications:

    Captioning: Figure 23

    Text generation: (Pretty bad, example from Karpathy's blog): "Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training.

12. Interestingly, some sequence problems can be addressed well with convolutional networks. e.g.,

    Yoon Kim - Convolutional Neural Networks for Sentence Classification - EMNLP - 2014

    WaveNet by Google for sound synthesis uses an "atrous convolution", which is a hierarchical structure. (Fig. 24)

13. See also http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/ http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

3.2 Sequence Learning

3.2.1 Motivation

1. Examples of problems involving sequence data
   - Speech recognition
   - Music generation
   - Time series forecasting

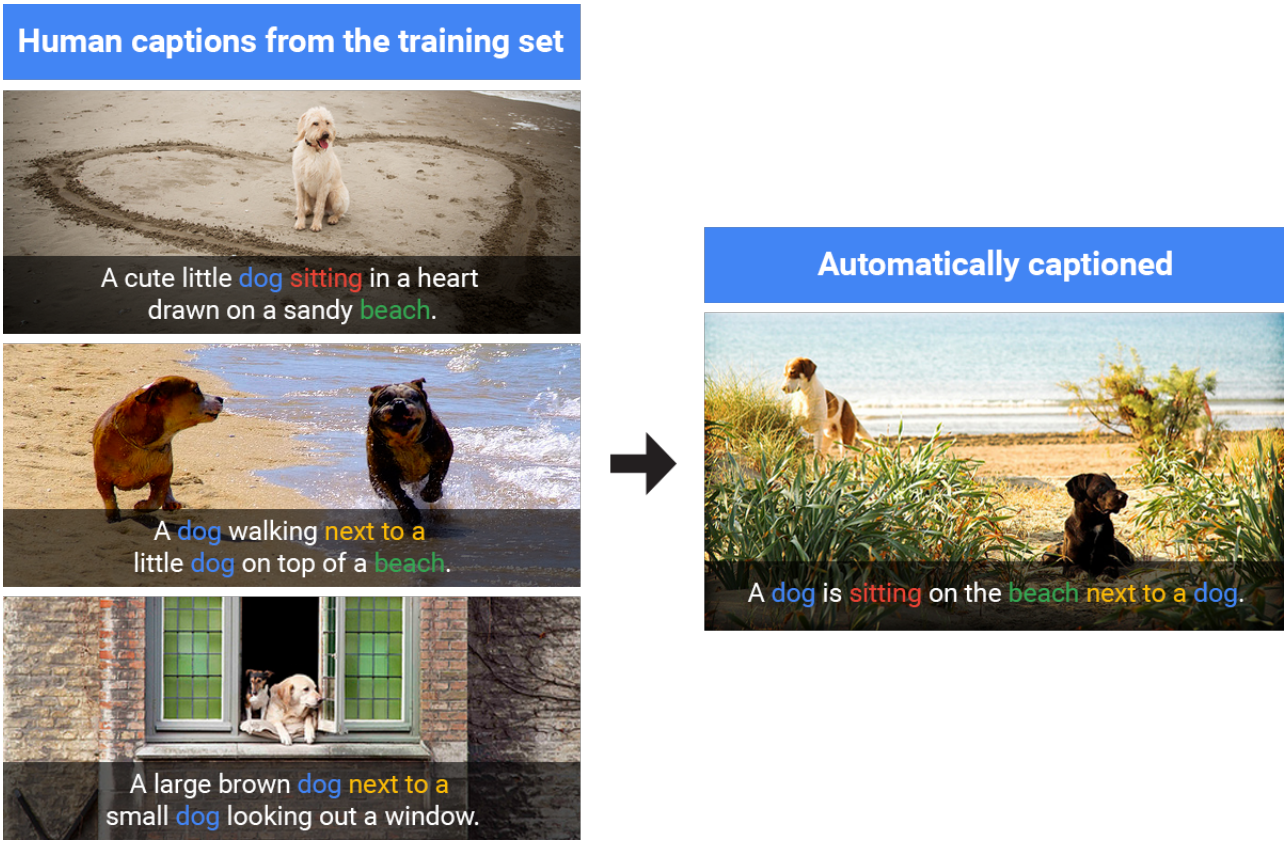


Figure 23: Captioning, Google Brain, 2016. Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge. https://ai.googleblog.com/2016/09/show-and-tell-image-captioning-open.html

- Machine translation
- Conversation agents
- Image captioning

2. Limitations of feedforward networks
   Recall the structure of a basic feedforward network: Figure 25

$$h = \sigma_1(b + Wx)$$
$$o = c + Vh$$
$$\hat{y} = \text{softmax}(o)$$

   Problem: How to learn from a sequence of inputs $x_1, x_2, \ldots, x_\tau$?
3. Sequence Models. Figure 26
   We will focus on the many-to-many cases. Requirements:

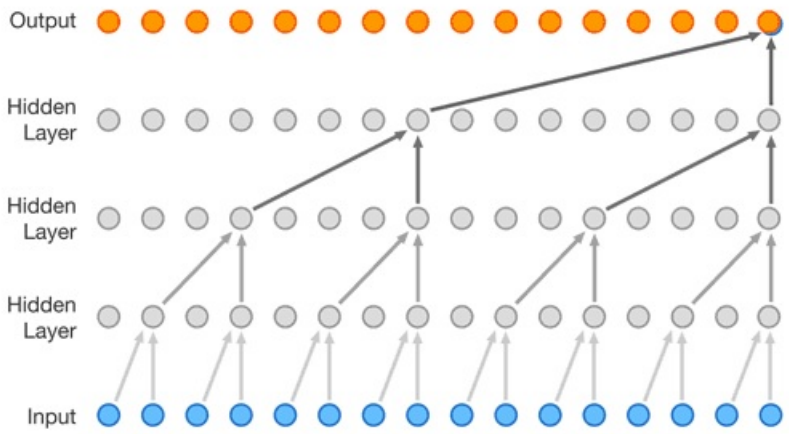


Figure 24: Wavenet.

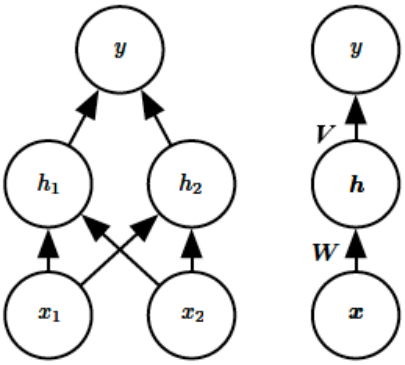


Figure 25: MLP.

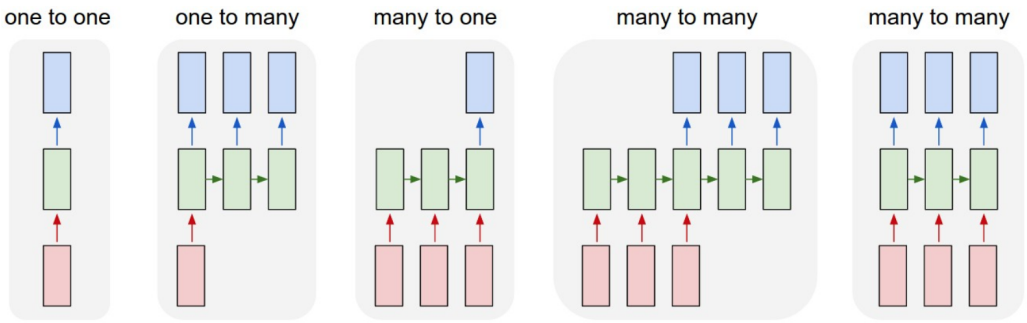


Figure 26: RNN types.

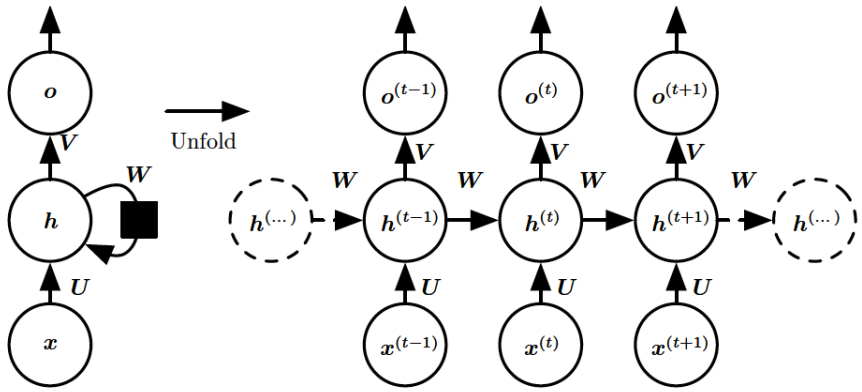


Figure 27: Simple RNN.

(a) Output $\hat{y}_t$ should depend on the sequence so far, $x_1, \ldots, x_t$.
(b) We may not know the length of a particular input sequence ahead of time.

3.3 Sequence Models

3.3.1 RNNs

1. A Simple Recurrent Neural Network. Figure 27

$$
\begin{aligned}
h_t &= \sigma_1(b + W h_{t-1} + U x_t) \\
o_t &= c + V h_t \\
\hat{y}_t &= \text{softmax}(o_t)
\end{aligned}
$$

2. A Simple Recurrent Neural Network
   RNNs allow us to learn a single model, rather than a separate one for each time step.
   (a) The model is specified in terms of *transitions* from one state h_t to the next.
   (b) Parameters are shared across time.

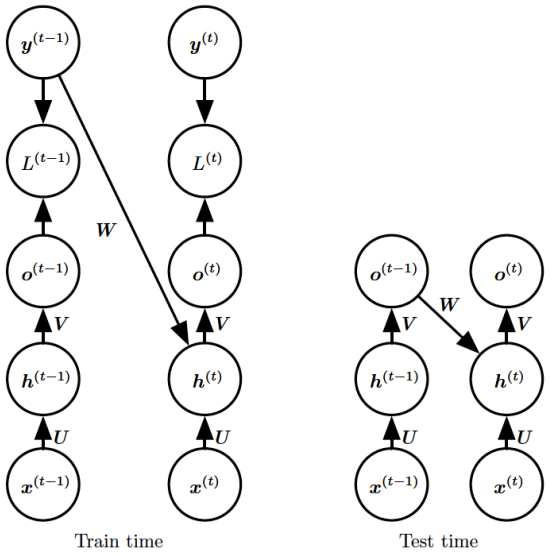


Figure 28: Teacher forcing.

Loss function is a sum of losses at each time step t:

$$L(\hat{y}, y) = \sum_{t=1}^{T} \ell(\hat{y}_t, y_t)$$

3. Training RNNs

   Training proceeds as before using **backpropagation through time** on the unrolled computation graph.

   We cannot easily parallelize training since each step is dependent on the one before it.

   Teacher forcing can be used when there are output-to-hidden recurrent connections. Ground truth is fed to the model instead of its own output. Figure 28

4. Modeling Joint Probability Distributions

   When we use a negative log-likelihood training objective,

   $$\ell(\hat{y}_t, y_t) = -\log \Pr(y_t \mid x_1, \ldots, x_t)$$

   we train the RNN to estimate the conditional distribution of the next sequence element y_t given the past inputs.

   We typically use the softmax function as the output layer to obtain normalized probabilities for each class.

   $$\text{softmax}(o)_i = \frac{e^{o_i}}{\sum_{j=1}^{\tau} e^{o_j}}$$

   RNNs can model arbitrary probability distributions of some sequence y over another sequence x.

   $$\Pr(y_1, \ldots, y_\tau \mid x_1, \ldots, x_\tau) = \prod_{t=1}^{\tau} \Pr(y_t \mid x_1, \ldots, x_t)$$

To remove the conditional independence assumption, we can add output-to-hidden connections.

$$\Pr(y_1, \ldots, y_\tau \mid x_1, \ldots, x_\tau) = \prod_{t=1}^{\tau} \Pr(y_t \mid y_1, \ldots, y_{t-1}, x_1, \ldots, x_t)$$

Some challenges:

(a) What if we want a given output y_t to depend on the entire sequence $x_1, \ldots, x_\tau$?

(b) How do we map a sequence x to a sequence y when they can differ in length from each other?

We will see the solutions later in the context of machine translation.

First, a fundamental problem:

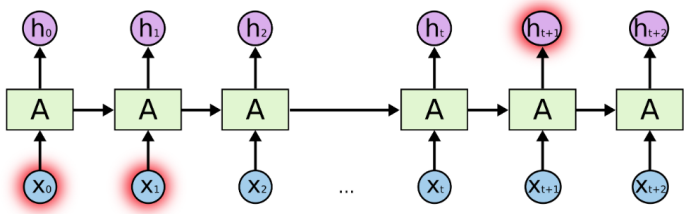
RNNs have trouble learning long-term dependencies.

3.3.2 Training Challenges

1. Learning Long-Term Dependencies

   Sentence 1: "Jane walked into the room. John walked in too. Jane said hi to..."

   Sentence 2: "Jane walked into the room. John walked in too. It was late in the day, and everyone was walking home after a long day at work. Jane said hi to..."



   RNNs have trouble learning dependencies from inputs with a large time difference from the predicted output.

2. Vanishing or Exploding Gradients

   Suppose we have an RNN with state h_t, input x_t, and cost $\mathcal{E}$.[1]

$$h_t = W\sigma(h_{t-1}) + Ux_t + b$$

$$\mathcal{E} = \sum_{1 \le t \le \tau} \mathcal{E}_t, \quad \mathcal{E}_t = \mathcal{L}(h_t)$$

   Let's calculate the gradient over one input sequence.

$$\frac{\partial \mathcal{E}}{\partial W} = \sum_{1 \le t \le \tau} \frac{\partial \mathcal{E}_t}{\partial W}$$

$$\frac{\partial \mathcal{E}_t}{\partial W} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial^+ h_k}{\partial W} \right)$$

$$\frac{\partial h_t}{\partial h_k} = \prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{t \ge i > k} W^\top \operatorname{diag}(\sigma'(h_{i-1}))$$

[1] The original paper uses this formulation instead of $h_t = \sigma(Wh_{t-1} + Ux_t + b)$ and says they are equivalent.

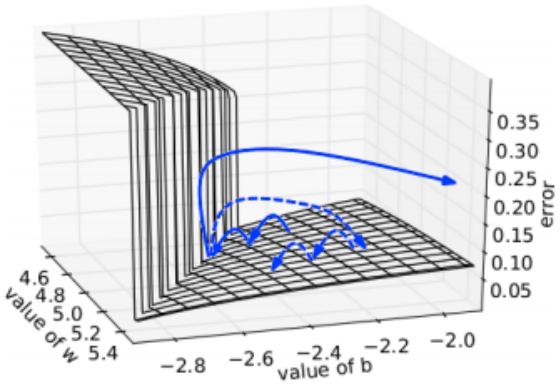


Figure 29: Gradient Clipping.

3. Vanishing or Exploding Gradients
   The cause is the Jacobian matrix J. We have a product of $t-k$ Jacobians.

$$J = \frac{\partial h_{k+1}}{\partial h_k} = W^\top \text{diag}(\sigma'(h_k))$$
$$\|J\| = \left\| \frac{\partial h_{k+1}}{\partial h_k} \right\| \leq \underbrace{\left\| W^\top \right\|}_{\lambda_1} \underbrace{\left\| \text{diag}(\sigma'(h_k)) \right\|}_{\gamma} = \lambda_1 \gamma$$

   λ_1 is the largest singular value of W. $\gamma = 1$ for tanh and $\gamma = \frac{1}{4}$ for sigmoid.

$$\left\| \prod_{i=k}^{t-1} \frac{\partial h_{i+1}}{\partial h_i} \right\| \leq (\lambda_1 \gamma)^{t-k}$$

   When $t \gg k$:
   - If $\lambda_1 < \frac{1}{\gamma}$, the gradients *will* vanish.
   - If $\lambda_1 > \frac{1}{\gamma}$, the gradients *may* explode.

4. How to deal with gradient problems?
   (a) Modify the training algorithm: **gradient clipping**.
      - Commonly used when training all variants of RNNs.
   (b) Use a different activation function: **ReLUs**.
      - Primarily used for deep neural nets (e.g. computer vision).
   (c) Use a more complex neural architecture: **LSTMs and GRUs**.

5. Gradient Clipping. Figure 29
   Addresses the problem of exploding gradients by preventing parameter updates from being too large.

   $g \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
   if $\|g\| \geq$ threshold **then**

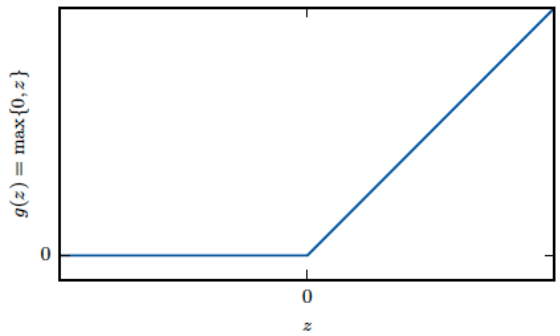


Figure 30: ReLU.

$$g \leftarrow \frac{\text{threshold}}{\|g\|} g$$

Does gradient clipping affect convergence?

6. ReLU. Figure 30

   Since the derivative is 1 when $z > 0$, gradients flow more easily compared to sigmoid or tanh units. Also computationally efficient.

   Can lead to "dead neurons" during training. Is this a problem?

   Empirically, ReLU has been shown to be very effective.

3.3.3 LSTM and GRU

1. New Recurrent Architectures

   We will introduce the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures.

   - Address the problem of vanishing gradients.
   - Considered the state of the art for many deep learning tasks.
     - Image captioning, parsing, speech recognition, machine translation, reinforcement learning

2. Long Short-Term Memory Network (Hochreiter 1997). Figure 31

$$\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
a_t &= \sigma_t(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= i_t \odot a_t + f_t \odot c_{t-1} \\
h_t &= o_t \odot \sigma_t(c_t)
\end{aligned}$$

3. LSTM and Backpropagating Gradients

   Recall the equation to update the cell state:

$$c_t = i_t \odot a_t + f_t \odot c_{t-1}$$

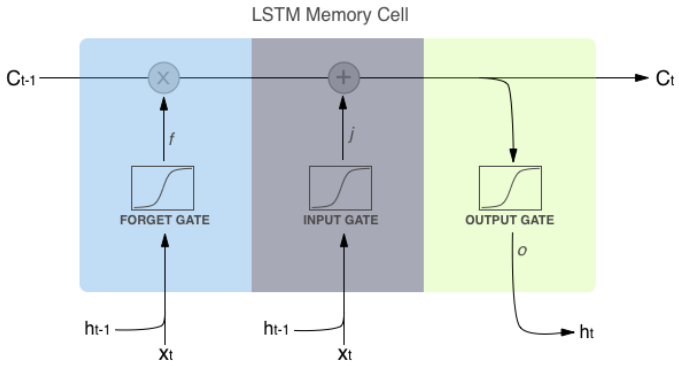


Figure 31: LSTM.

The original LSTM did not have a forget gate, allowing error to flow unchanged from c_t to c_{t-1}.

- Referred to as the constant error carousel.

With a forget gate, if $f_t \approx 1$, we achieve the same effect.

- Can initialize the forget gate bias to 1 before training.

4. Gated Recurrent Unit (Cho et. al 2014)

$$
\begin{aligned}
u_t &= \sigma(W_u x_t + U_u h_{t-1} + b_u) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
a_t &= \sigma_t(W_i x_t + r_t \odot U_i h_{t-1} + b_i) \\
h_t &= u_t \odot h_{t-1} + (1 - u_t) \odot a_t
\end{aligned}
$$

Hidden units that learn to capture...

- short-term dependencies will tend to have reset gates that are frequently active.
- longer-term dependencies will have update gates that are mostly active.

Takeaway: Similar performance to LSTM, but fewer parameters.

3.4 Machine Translation

1. Neural Machine Translation in 2016. Figure 32
2. History

   A bilingual translation task:

   "The cat sat on the mat." → "Le chat s'est assis sur le tapis."

   * * *

   2003: Neural language model introduced by Bengio et al. This was incorporated into existing phrase-based statistical machine translation (SMT) systems.

   2014: Encoder-decoder RNN introduced by Cho et. al for use in SMT.

   2015: Attention model proposed by Bahdanu et. al for end-to-end neural machine translation (NMT).

   2017: Transformer model proposed by Vaswani et. al. This is the current state-of-the-art.

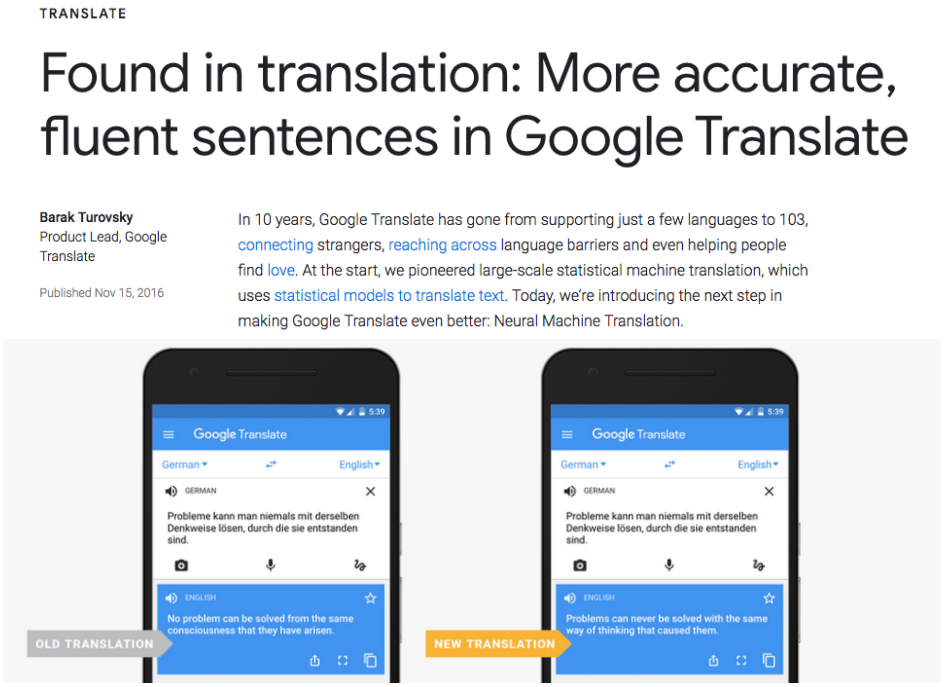


Figure 32: Neural Machine Translation in 2016.

3. Word Representations

   How should the model encode a word token in a sequence?

   Attempt 1: One-hot vectors. Represent every word as an $\mathbb{R}^{|V|\times 1}$ vector with all zero's except for a single one depending on the index of the word in the vocabulary V.

   - $e(\text{cat}) = [\,0\ 0\ 0\ \ldots\ 1\ \ldots\ 0\,]^\top$
   - Does not capture semantic similarity between words!
   - Scales poorly with size of vocabulary.

4. Word Representations

   Attempt 2: Word embeddings. Model each word in a low-dimensional continuous space with dimension M.

   - $e(\text{cat}) = [\,0.1\ 0.33\ 0.72\ \ldots\ 0.59\,]^\top$
   - Intuition: "A word is defined by the company it keeps."

   Has revolutionized natural language processing (NLP) tasks since 2010.

   - Popular word embeddings: word2vec, GloVe, ELMo

5. Encoder-Decoder Model

   How to map an input sequence to an output sequence where the lengths are not necessarily the same?

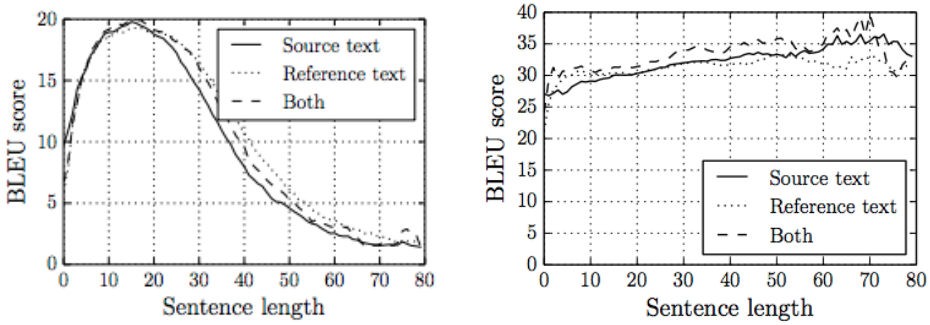


Figure 33: Translation quality vs. sentence length. RNN left, SMT right.

Source	She explained her new position of foreign affairs and security policy representative as a reply to a question: "Who is the European Union? Which phone number should I call?"; i.e. as an important step to unification and better clarity of Union's policy towards countries such as China or India.
Reference	Elle a expliqué le nouveau poste de la Haute représentante pour les affaires étrangères et la politique de défense dans le cadre d'une réponse à la question: "Qui est qui à l'Union européenne?" "A quel numéro de téléphone dois-je appeler?", donc comme un pas important vers l'unicité et une plus grande lisibilité de la politique de l'Union face aux états, comme est la Chine ou bien l'Inde.
RNNEnc	Elle a décrit sa position en matière de politique étrangère et de sécurité ainsi que la politique de l'Union européenne en matière de gouvernance et de démocratie .
grConv	Elle a expliqué sa nouvelle politique étrangère et de sécurité en réponse à un certain nombre de questions : "Qu'est-ce que l'Union européenne ? " .
Moses	Elle a expliqué son nouveau poste des affaires étrangères et la politique de sécurité représentant en réponse à une question: "Qui est l'Union européenne? Quel numéro de téléphone dois-je appeler?"; c'est comme une étape importante de l'unification et une meilleure lisibilité de la politique de l'Union à des pays comme la Chine ou l'Inde .

Figure 34: Sample output of two RNN models compared to the SMT system, Moses.

Encoder RNN processes the input sequence and emits a context vector c, which is the model's final hidden state, h_{τ_x}.

$$h_t = f(h_{t-1}, e(x_t))$$

Decoder RNN generates the output sequence based on c and the previous output.

$$s_t = f(s_{t-1}, e(y_{t-1}), c)$$
$$\Pr(y_t) = g(s_t, e(y_{t-1}), c)$$

Jointly trained to maximize $\log \Pr_\theta(y_1, \ldots, y_{\tau_y} \mid x_1, \ldots, x_{\tau_x})$.

6. Figure 33

   Task: English-to-French translation.

   Training data: 348M sentences from Europarl, news commentary, etc.

   Test data: 3000 sentences from a standard newstest dataset.

7. Figure 34

   Problem: the neural network must compress all information in the source sentence into a single, fixed-length vector.

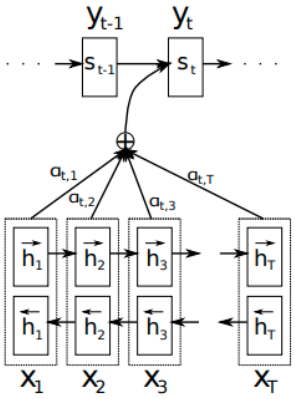


Figure 35: Attention

3.4.1 Attention

1. Encoder-Decoder Model with Attention

   Decoder RNN is similar to before, but each context vector c_t is distinct for each target word y_t:

   $$s_t = f(s_{t-1}, e(y_{t-1}), c_t)$$
   $$\Pr(y_t) = g(s_t, e(y_{t-1}), c_t)$$

   c_t is a weighted sum of *annotations* $h_1, \ldots, h_{\tau_x}$ to which the encoder maps the input sentence.

   $$c_t = \sum_{j=1}^{\tau_x} \alpha_{tj} h_j$$

   How are α and h computed?

2. The encoder is a "**bidirectional RNN**."

   Forward RNN reads the input as ordered and computes a sequence of forward hidden states $\overrightarrow{h}_1, \ldots, \overrightarrow{h}_{\tau_x}$.

   Backward RNN reads the input in reverse and computes a sequence of backward hidden states $\overleftarrow{h}_1, \ldots, \overleftarrow{h}_{\tau_x}$.

   An annotation for a word x_j is the concatenation of the two hidden states.

   $$h_j = \left[\overrightarrow{h}_j, \overleftarrow{h}_j\right]$$

   Due to the tendency of RNNs to better represent recent inputs, the annotation h_j will be focused on the words around x_j.

3. Encoder-Decoder Model with Attention. Figure 35

   To compute the weight α_{ij} of each annotation h_j, we use an **alignment model**. This is just a single-layer feedforward NN.

   $$e_{ij} = a(s_{i-1}, h_j)$$
   $$\alpha_{ij} = \text{softmax}(e_{ij})$$

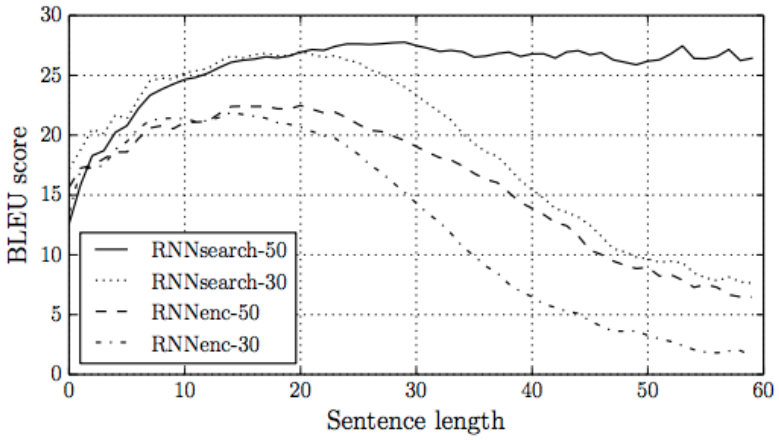


Figure 36: Encoder-Decoder Model with Attention.

The alignment model scores how well the inputs around position j and the output at position i match.

It is interesting that the alignment refers to representations that are learned from the data. Namely, the annotations h_j and the Decoder states s_t

4. Figure 36

   Same training and test data as before (English-to-French).

   The encoder-decoder model with attention does much better on longer sentences.

5. Figure 37

3.4.2 Transformer Model

1. Transformer Model

   "Attention is All You Need"

   * * *

   The transformer model utilizes an encoder/decoder architecture with attention, but no recurrent connections!

   Recall that training RNNs is not parallelizable and requires more memory for each example due to the recurrence.

   Also achieves state-of-the-art performance on translation tasks.

2. NMT Model Comparison (December 2017). Figure 38

   Self-attention allows the Transformer to be trained in a more parallel fashion on GPU hardware. Typically, $d > n$.

3. Transformer Model. Figure 39

   No recurrent connections: the entire input sequence/output sequence so far is sent into the model at once (demo).

   (a) Input words get transformed into "positional embeddings."

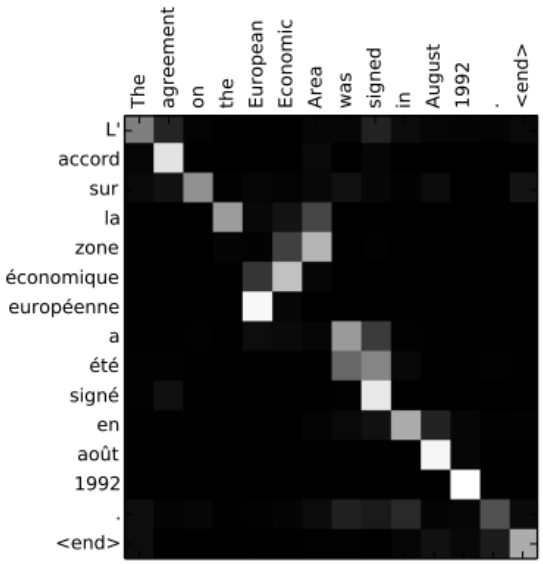


Figure 37: A sample alignment found by RNNsearch-50.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Figure 38: n is sequence length, d is representation dimension, k is kernel size of convolutions, r is the size of the neighborhood in restricted self-attention.

(b) For each encoder layer:
   i. Apply multi-headed self-attention.
   ii. Send outputs through feedforward network.

(c) For each output word:
   i. Transform all previous output words into positional embeddings.
      A. Apply masked multi-headed self-attention.
      B. Apply attention using the encoder outputs.
      C. Send output through feedforward network.
   ii. Transform output vector into the next word using linear and softmax layers.

4. Self-Attention. Figure 40

   Intuition: When the model processes a word, self-attention allows it to look at other positions in the input sequence to help it determine the optimal encoding for the word.

   Example: "The animal didn't cross the street because it was too tired."

   - Visualization

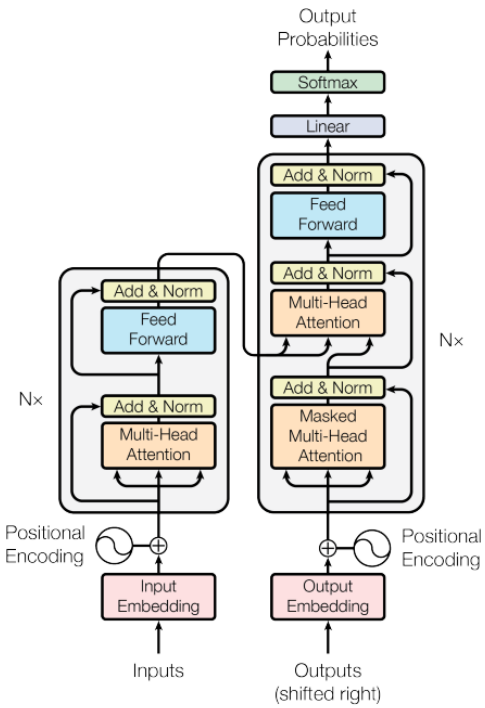


Figure 39: Transformer Model.

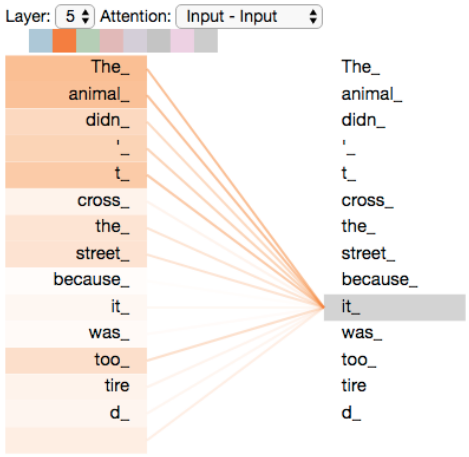


Figure 40: Self-Attention.

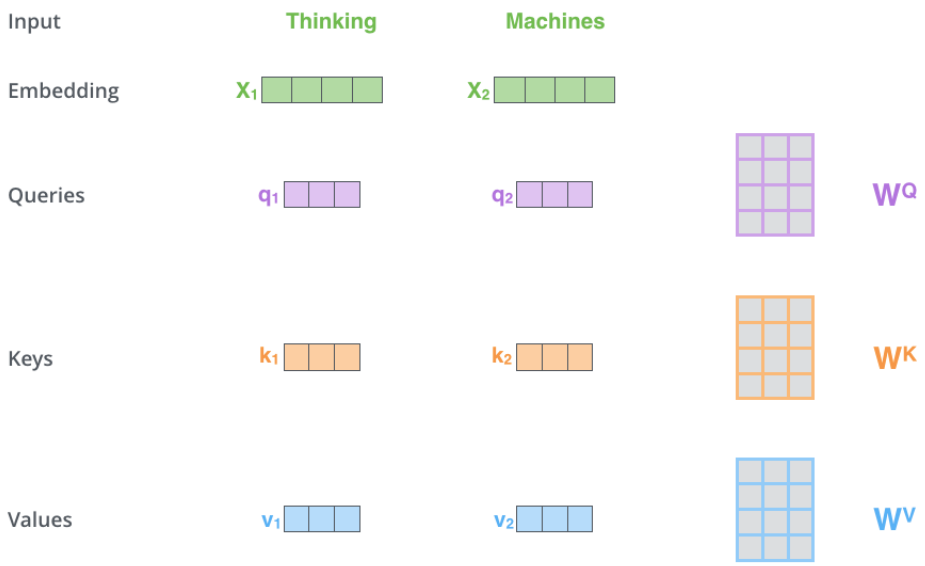


Figure 41: Self-Attention 1.

5. Self-Attention. Figure 41

   Step 1: Generate query, key, and value vectors for each embedding using learned matrices W^Q, W^K, W^V.

6. Figure 42

   Step 2: Compute a score for each key-value pair. Normalize these weights to sum to 1, and compute the output as the weighted sum of the values.

7. Figure 43

   Multi-headed attention means learning M attention layers in parallel (with different weight matrices).

   Step 3 is combining the results using another learned matrix W^O.

8. NMT Model Comparison (December 2017). Figure 44

9. Future Directions for NMT

   Explore attention and CNNs as an alternative to RNNs.

   Many active research areas in NMT, including:

   (a) Sub-word or character-level models.
   (b) Rare word problem.
   (c) Low-resource languages.
   (d) Efficient decoding.

10. References

    Pascanu, Mikolov, Bengio (2013) On the difficulty of training recurrent neural networks *arXiv preprint* arXiv:1211.5063

    Cho, Bahdanau, Bougares, Schwenk, Bengio (2014) Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation *arXiv preprint* arXiv:1406.1078

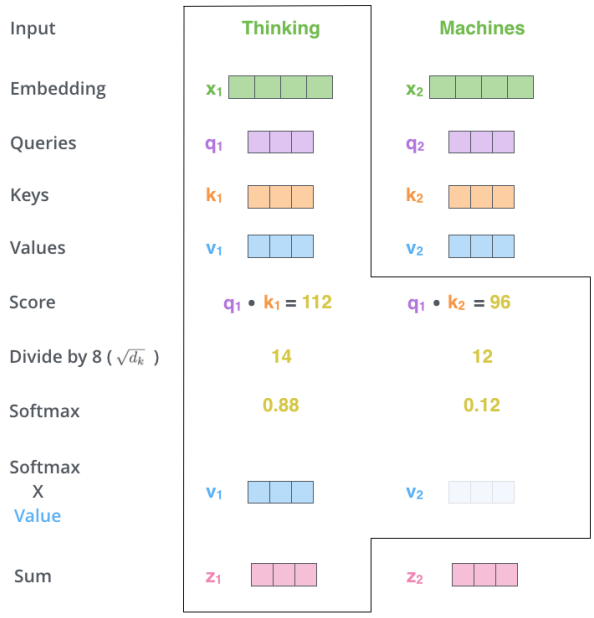


Figure 42: Self-Attention 2

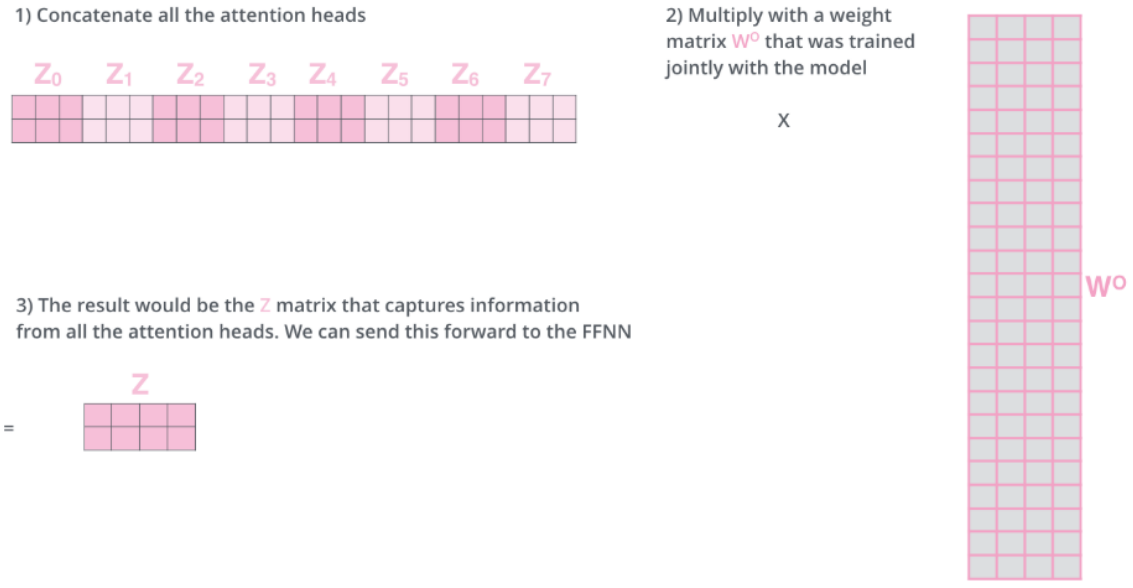


Figure 43: Self-Attention 3

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Figure 44: The Transformer outperforms other state-of-the-art models at a fraction of the training cost. FLOPS is floating point operations.

Bahdanau, Cho, Bengio (2015) Neural Machine Translation by Jointly Learning to Align and Translate *arXiv preprint* arXiv:1409.0473

Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, Polosukhin (2017) Attention Is All You Need *arXiv preprint* arXiv:1706.03762

3.5 Image + Text

1. Models that combine images and text have a special feel of "intelligence". We are used to models for classifying images and translating text, but generating captions for an image, or answering questions aboout it, may be more unusual. This is in addition to the obvious practical importance of these tasks.
2. Karpathy, Fei-Fei Li, Deep Visual-Semantic Alignments for Generating Image Descriptions
   (a) Task: generate natural language descriptions of different image regions. Input image, output Regions+Captions. Figure 45
   (b) Data: image + sentence. These are weak labels (not segmented)
   (c) Alignment: input image and a sentence, output: score for how well they match
   Segment: R-CNN. Map each region to 500-dimensional space.
   Text: bidirectional RNN. Map each word of sentence to 500-dimensional space.
   Can compute similarity/inner product/alignment.
   "since we are ultimately interested in generating snippets of text instead of single words, we would like to align extended, contigu- ous sequences of words to a single bounding box. To address this issue, we treat the true alignments as latent variables in a Markov Random Field (MRF) where the binary interactions between neighboring words encourage an alignment to the same region"
   (d) "Multimodal Recurrent Neural Network for generating descriptions". Generation: Input: image. Output: Embedding.
   Feed to RNN as a state variable. Output: Caption.
   [Amazing that you can represent text and images in the same space.]
3. Visual Question Answering (Q+A)
   (a)

Figure 45: Deep Visual-Semantic Alignments for Generating Image Descriptions

4 Unsupervised learning

4.1 Setup

1. Unsupervised learning: we have features x_i, but no outcomes y_i. We are trying to understand structure, summarize, compress, denoise, visualize, the data.

   This data can be easier to collect. However, it can be more removed from the "end-goal" or "decision-making". In supervised learning we can make a yes/no decision that we can directly act on. In unsupervised learning, we are often making an intermediate decision.

   Most direct economic value comes from supervised learning, where we predict outcomes.

2. Classical statistics and statistical learning:

   Non-parametric density estimation: kernel methods, local polynomial methods

   Dimension reduction: PCA, ICA, random projections

   Clustering: k-means, EM

3. Connections with supervised learning: There is a rich set of connections to supervised learning.

   Reductions: Often unsupervised methods can be viewed as reductions to supervised learning. e.g., in dimension reduction, we try to find a function f of the data (in some class), so that there is some other function g of the data (in some class), so that $g(f(X))$ approximates X well.

   Semi-supervised Learning: Given a (small) amount of labeled data and a (large) amount of unlabeled data, learn predictor. Example: learning parse trees, image search, ...

   Self-supervised learning: Construct your own supervision. For instance, predict some part of the image. Predict the next frame in a video. Yann LeCun advocates that this is important, because "the future will not be supervised".

4.2 PCA, AE, VAE

1. PCA. Widely used for exploratory data analysis

   Data matrix X, $n \times p$, has p features of n samples (e.g., genomic features of samples from Europe). Center each column of X.

   Principal Component Analysis (PCA): Find linear combinations

   $$f = Xv = \sum_i v_i X_i$$

   of features that maximize variance.

   Sequentially, subject to orthogonality constraints. First PC: maximize empirical variance

   $$\max_v \ \text{Var}\,[Xv] = n^{-1} v^\top X^\top X v$$
   $$s.t.\ v^\top v = 1$$

   Nonconvex optimization problem. Solution v: top eigenvector of sample covariance matrix $\widehat{\Sigma} = n^{-1} X^\top X$.

   Variance: largest eigenvalue of $\widehat{\Sigma}$. Similarly at all steps: PC scores - eigenvectors, variances - eigenvalues.

   Equivalently, can write all steps in one as

   $$\max_V \ \text{tr}(V^\top X^\top X V)$$
   $$s.t.\ V^\top V = I_k$$

   or

   $$\min_V \ \|X(I - VV^\top)\|_{Fr}^2$$
   $$s.t.\ V^\top V = I_k$$

   This last reformulation can be viewed as a reduction to supervised learning. We try to find a function $Z := f(X) = XV$ of the data that reduces the dimension from p to k, so that the pseudoinverse $g(Z) = ZV^\top$ approximates X well. In this case, we impose the constraint that $VV^\top$ is an orthogonal projection matrix. However, if we are looking for a linear subspace (manifold), that best approximates the data, we automatically get that the approximation operation has to be a linear projection. So, effectively, PCA = "Best Linear Subspace."

2. Autoencoders (AE): Relax the linear maps to more general maps, e.g., neural networks. Figure 46,

   $$\min \ \sum_i \|x_i - g(z_i, W_g)\|^2$$
   $$s.t.\ z_i = f(x_i, W_f)$$

   Note: Encoder and Decoder

   Training: backprop

   Empirical results?

   Probably not good at generating new meaningful images.

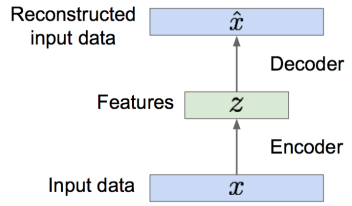


Figure 46: AE

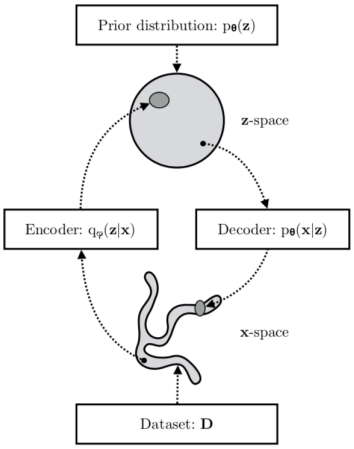


Figure 47: VAE

3. Variational Autoencoders (VAE). Figure 47,

   Instead, model the distribution of the latent variable z probabilistically, and try to fit models for the probability distributions $p(x|z), q(z|x)$.

   Parametrize:

   (a) Prior distribution $p_\theta(z)$.

   (b) Conditional distribution $p_\theta(x|z)$.

   High-level goal: maximize (marginal) likelihood.

$$\max_\theta \sum_i \log p_\theta(x_i)$$

   Express in terms of latent variable z

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

   In principle, if we maximize over θ, then theoretically we obtain the posterior distribution as $p(z|x) = p(x|z)p(z)/p(x)$. How can we train this model?

   The direct approach has issues. We need to approximate the z-integral by a quadrature, which can become prohibitive in high dimensions.

4. Different approach: The parameters of the VAE can be estimated efficiently in the stochastic gradient variational Bayes (SGVB) framework, where the variational lower bound of the log-likelihood is used as a surrogate objective function.

   First we introduce and parametrize an approximation to the posterior $p(z|x)$:

   (a) Conditional distribution $q_\phi(z|x)$.

   See Figure 48 for the sampling process of VAEs:

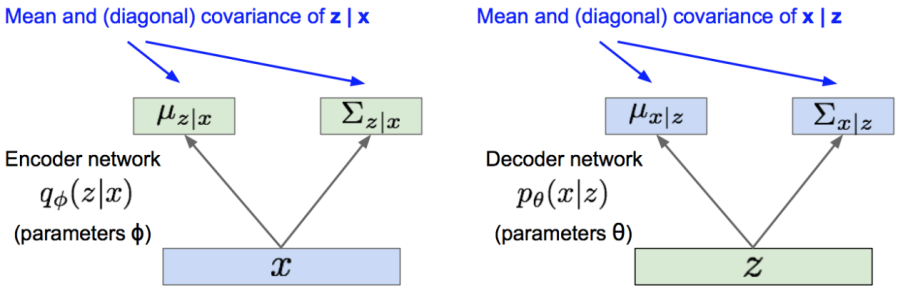


Figure 48: Sampling process of VAEs.

The variational lower bound is written as:

$$\log p_\theta(x) \geq \mathbb{E}_z[\log p_\theta(x|z)] - \mathbb{KL}[q_\phi(z|x)||p_\theta(z)]$$

5. Here are the details. We want to maximize the data likelihood:

$$p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$$

$$\begin{aligned}
\log p_\theta(x) &= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x)] \\
&= \mathbb{E}_z[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}] \\
&= \mathbb{E}_z[\log \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)} \frac{q_\phi(z|x)}{q_\phi(z|x)}] \\
&= \mathbb{E}_z[\log p_\theta(x|z)] - \mathbb{E}_z[\log \frac{q_\phi(z|x)}{p_\theta(z)}] + \mathbb{E}_z[\log \frac{q_\phi(z|x)}{p_\theta(z|x)}] \\
&= \mathbb{E}_z[\log p_\theta(x|z)] - \mathbb{KL}[q_\phi(z|x)||p_\theta(z)] + \mathbb{KL}[q_\phi(z|x)||p_\theta(z|x)]
\end{aligned}$$

The last KL-Divergence is greater than or equal to 0. Define:

$$\mathcal{L}(x, \theta, \phi) = \mathbb{E}_z[\log p_\theta(x|z)] - \mathbb{KL}[q_\phi(z|x)||p_\theta(z)]$$

Then $\log p_\theta(x) \geq \mathcal{L}(x, \theta, \phi)$. In VAEs we try to maximise the lower bound

$$\sum_{i=1}^{n} \mathcal{L}(x_i, \theta, \phi)$$

We can try to approximate the expectations by sampling. Both expectations are over z, so we can sample several independent z_j (using the current paramters ϕ_t) and use plug-in estimators for the expectations.

$\mathbb{E}_z[\log p_{\theta_t}(x|z)] \approx m^{-1} \sum_{j=1}^m \log p_{\theta_t}(x|z_j)$

$\mathbb{KL}[q_{\phi_t}(z|x)||p_{\theta_t}(z)] \approx m^{-1} \sum_{j=1}^m \log \frac{q_{\phi_t}(z_j|x)}{p_{\theta_t}(z_j)}$

We do this for all $x = x_i$, and take gradient steps with respect to ϕ, θ. The problem with this is that it does not give the correct gradient with respect to ϕ, as the expectation depends on ϕ and we have sampled according to that, so the derivative of the sampling will not give the dependence of the probability measure on ϕ

6. Reparameterization trick. The key to the problem is the reparametrization trick, which evaluates z as a deterministic function of ϕ as $z = g(\phi, x, \varepsilon)$ instead of sampling from a probability distribution parametrized by ϕ. Then we can easily evaluate the derivative wrt ϕ by backprop. See Figure 49

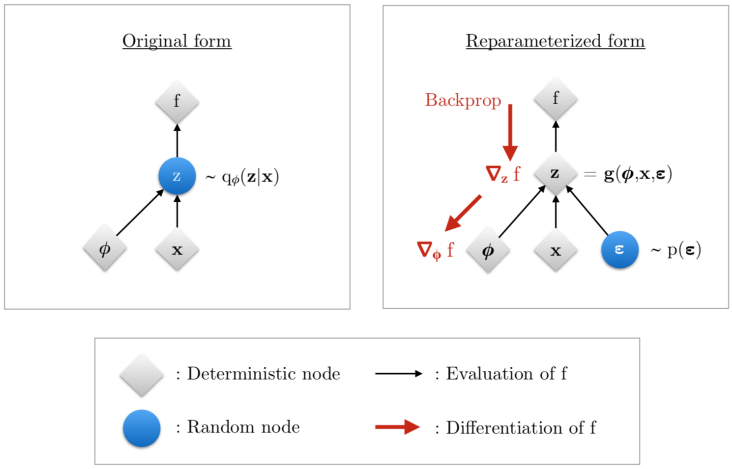


Figure 49: Reparameterization trick for VAEs.

Putting it all together: See Figure 50

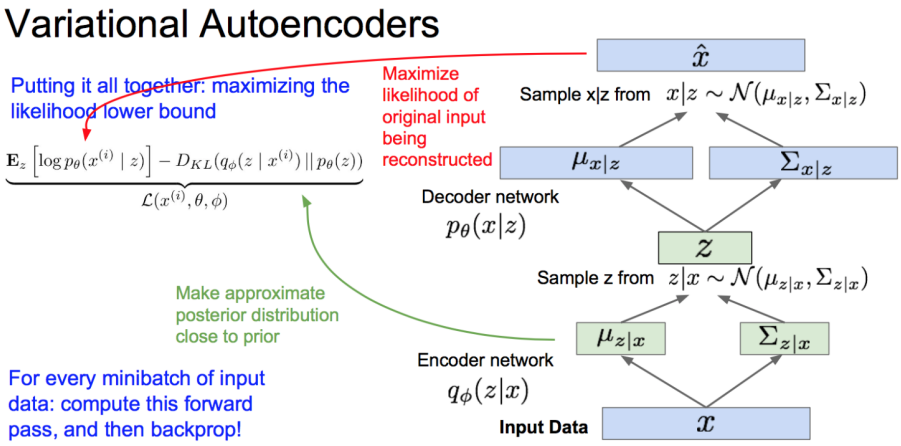


Figure 50: Framework of VAEs.

7. Application on Faces:

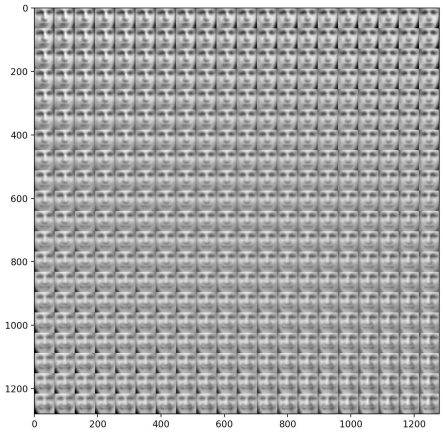


Figure 51: Generated faces

https://distill.pub/2017/aia/

4.3 Generative adversarial networks (GAN)

1. *Generative adversarial networks (GANs)* are a method proposed first for unsupervised learning (Goodfellow et al, 2014). Considered by some experts to be the most important development in machine learning in the last 20 years.

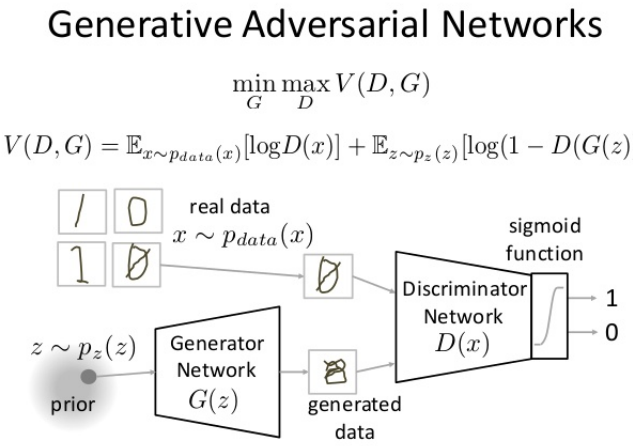


Figure 52: GAN

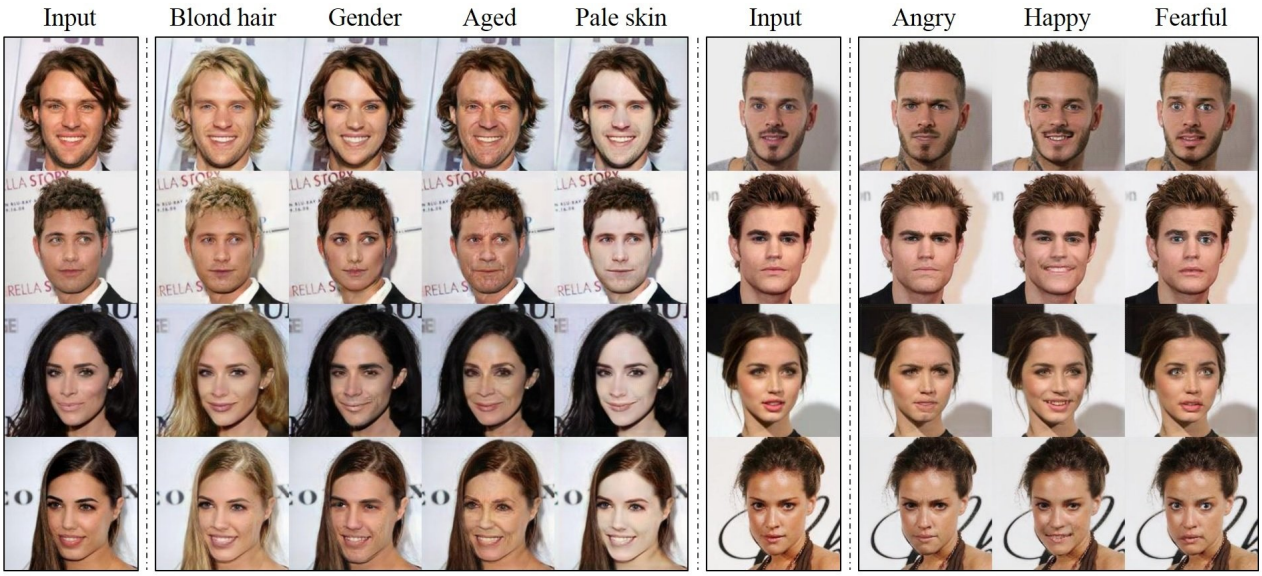


Figure 53: GAN Example

2. Example results: Figures 53, 54
3. Want to learn density of images. In classical statistics, one would approach this problem by fitting a density P_θ from a given family (parametric or non-parametric) to the empirical data distribution.

   GANs learn a density by learning a *generator* G and a *discriminator* D. The generator is an algorithm to sample from the density of images, while the discriminator is an algorithm

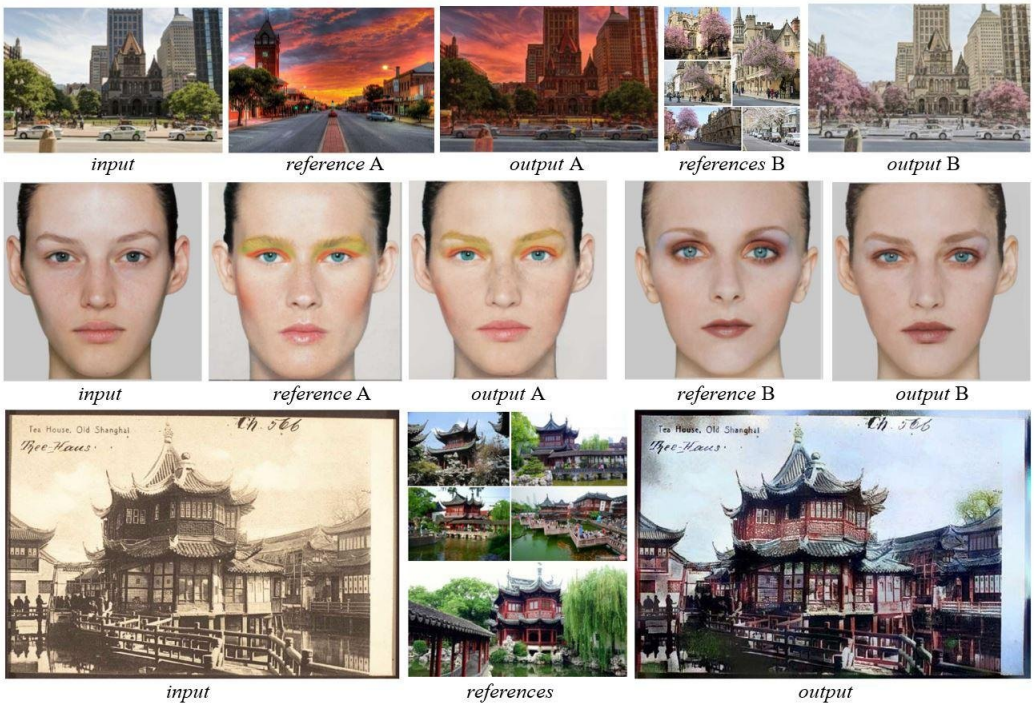


Figure 54: GAN Example

to distinguish the real from the generated images. By simultaneously training the two *adversaries*, we end up with both a good classifier and a good generator. (Figure 52)

4. More formally, let
   - P_n: data distribution
   - G: a generator that takes in a random vector Z, and produces a generated sample $G(Z) \sim P_G$
   - D: a discriminator that takes in a datapoint, and is supposed to return close to 1 for real data, and close to 0 for generated data (aka classifier)

   The original GAN objective:

$$\min_G \max_D V(D,G) = \mathbb{E}_{P_n} \log D(X) + \mathbb{E}_G \log[1 - D(G(Z))]$$

   The reason why these are called networks is that G, D are both taken to be neural networks
5. Training initializes both networks randomly, and then performs *alternating gradient updates* as follows. Iterate:
   (a) Generate a few samples $G(Z_i)$ from the current density (say m samples, and call the distribution G_k)
   (b) Update the discriminator parameters using a gradient step on the generated and real data (this update depends on the current generator)
   Specifically, the objective is

$$\hat{V}(G) = \mathbb{E}_{P_n} \log D(X) + \mathbb{E}_{G_k} \log[1 - D(G(Z))]$$
$$= n^{-1} \sum_{i=1}^{n} \log D(X_i) + m^{-1} \sum_{j=1}^{m} \log[1 - D(G(Z_j))]$$

We compute the gradient of this wrt the parameters θ of the net D via backprop.

(c) Update the generator using a gradient step to better match the true density (the gradient step now depends on the current discriminator)

6. Some basic results:

Proposition (Goodfellow et al, 2014). For a fixed generator G, the *optimal discriminator* D has the form

$$D^*(x) = \frac{p_n(x)}{p_n(x) + p_G(x)},$$

where p_n is the data density, and p_G is the generated density. The optimal discriminator can be defined arbitrarily outside of the support of the two densities.

Consider next the *equilibrium value* of the game, i.e., the solution to the minimax problem:

$$\min_G C(G) = V(D^*, G)$$

One can check $G(G) = 2JS(P_n \| P_G) - \log(4)$, where JS is the *Jensen-Shannon divergence*,

$$JS(P\|Q) = \frac{1}{2} KL(P\|(P+Q)/2) + \frac{1}{2} KL(Q\|(P+Q)/2).$$

Recall that the *Kullback-Leibler (KL) divergence* is

$$KL(P\|Q) = \mathbb{E}_P \log(P(X)/Q(X)).$$

Therefore, the original GAN can be viewed as minimizing

$$\min_G JS(P_n \| P_G)$$

By the well-known properties of JS divergence, within the class of distributions that have a density wrt P_G, this is minimized at P_G.

7. Now we can see that this is just a form of risk minimization. Later work considered other versions, motivated by the numerical instability of the original GANs. For instance *Wasserstein GAN (WGAN)* (Arjovsky et al, 2017) solves

$$\min_G d(P_G, P_n) = \min_G \max_{F \in \mathcal{F}} [\mathbb{E}_{P_G} F - \mathbb{E}_{P_n} F]$$

Here $d(P, Q)$ is the *Wasserstein-1 metric*, or *earth-mover distance*, and $\mathcal{F}$ is the class of 1-Lipshitz functions defined on the domain S where all probability distributions reside:

$$\mathcal{F} = \{F : S \to \mathbb{R} : |F(X) - F(Y)| \leq \|X - Y\|\}$$

To solve, we follow the same alternating gradient scheme as before. Iteratively,

(a) Update F

(b) Update P

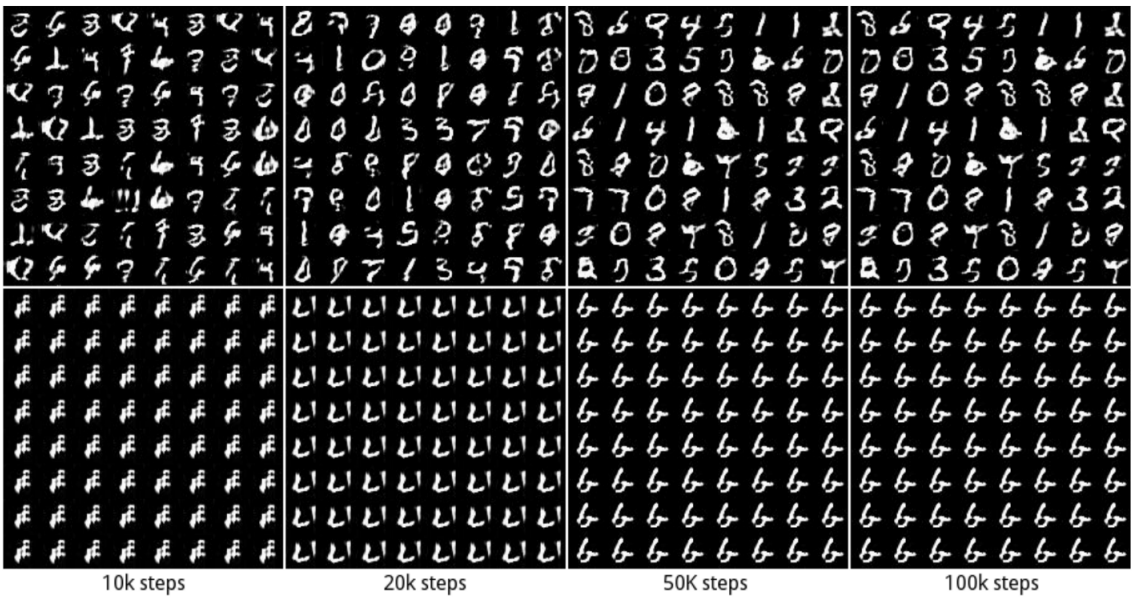


Figure 55: Mode collapse. Top: "Unrolled GAN", bottom: standard GAN

Approximate $\mathbb{E}$ by sampling.

[Justify]

8. *Stability?* Training GANs is notoriously unstable. How can they be made stable?

   Hacks: https://github.com/soumith/ganhacks

9. *Mode collapse.* GANs often suffer from a form of underfittign known as mode collapse. Specifically, they drop modes of a distribution, and generate only from the rest. Figure 55.

10. GauGAN

4.4 Wasserstein GAN

1. Notation. Let $\mathcal{X}$ be a compact metric space. Let Σ denote all Borel sets of $\mathcal{X}$. Let $Prob(\mathcal{X})$ denote the space of probability measures defined on $\mathcal{X}$. Now we define some elementary distances and divergences between two distributions $P_r, P_g \in Prob(\mathcal{X})$

2. The *Total Variation* (TV) distance is

$$\delta(P_r, P_g) = \sup_{A \in \Sigma} |P_r(A) - P_g(A)|$$

3. The *Kullback-Leibler* (KL) divergence is

$$KL(P_r \| P_g) = \int_{\mathcal{X}} \log(\frac{dP_r}{dP_g}) dP_r$$

   Note that P_r needs to be absolutely continuous with respect to P_g. Actually the KL divergence is not a distance since it is asymmetric.

4. The *Jensen-Shannon* (JS) divergence is

$$JS(P_r, P_g) = KL(P_r \| P_m) + KL(P_g \| P_m)$$

where P_m is $\frac{P_r + P_g}{2}$. This divergence is symmetric and always well-defined.

5. The *Earth-Mover* (EM) distance or Wasserstein-1 distance is

$$W(P_r, P_g) = \inf_{\gamma \in \prod(P_r, P_g)} E_{(x,y)\sim\gamma}[\|x - y\|]$$

where $\prod(P_r, P_g)$ denote the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively P_r and P_g. Intuitively, $\gamma(x, y)$ indicates how much "mass" must be transported form x to y in order to transform the distribution P_r into the distribution P_g. The EM distance the is the "cost" of the optimal transport plan.

6. *Example* (learning parallel lines). Let $Z \sim U[0, 1]$ the uniform distribution on the unit interval. P_0 denotes the distribution of $(0, Z) \in R^2$ and P_θ denotes the distribution of $(\theta, Z) \in R^2$. Then:

   - $\delta(P_0, P_\theta) = \begin{cases} 1 & if\ \theta \neq 0, \\ 0 & if\ \theta = 0 \end{cases}$

7.
   - $KL(P_0, P_\theta) = KL(P_\theta, P_0) = \begin{cases} \infty & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0 \end{cases}$
   - $JS(P_0, P_\theta) = \begin{cases} \log(2) & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0 \end{cases}$
   - $W(P_0, P_\theta) = |\theta|$

8. *Theorem 1.* Let P_r be a fixed distribution over $\mathcal{X}$. Let Z be a random variable (e.g. Gaussian) over another space $\mathcal{Z}$. Let $g : \mathcal{Z} \times R^d \to \mathcal{X}$ be a function, denoted $g_\theta(z)$. Let P_θ denote the distribution of $g_\theta(Z)$. Then:
   1. If g is continuous in θ, so is $W(P_r, P_\theta)$.
   2. If g is locally Lipschitz and satisfies some regularity assumption, then $W(P_r, P_\theta)$ is continuous everywhere, and differentiable almost everywhere.
   3. Statements 1-2 are false for the JS divergence $JS(P_r, P_\theta)$

9. Remarks.

   (a) $g_\theta(z)$ is locally Lipschitz iff for a given pair (θ, z) there is a constant $L(\theta, z)$ and an open set U s.t. for every $(\theta', z') \in U$ we have

$$\|g_\theta(z) - g_{\theta'}(z')\| \leq L(\theta, z)(\|\theta - \theta'\| + \|z - z'\|)$$

   (b) suppose $g_\theta(z)$ is locally Lipschitz, then we say that $g_\theta(z)$ satisfies the *regularity assumption* for a certain probability distribution p over $\mathcal{Z}$ if the local Lipschitz constants $L(\theta, z)$ satisfy $E_{z\sim p}[L(\theta, z)] < +\infty$

10. *Corollary 1.* Let g_θ be any feedforward neural network parametrized by θ, and $p(z)$ a prior over z such that $E_{z\sim p(z)}[\|z\|] < \infty$(e.g. Gaussian, uniform, etc) then $g_\theta(z)$is locally Lipschitz and satisfies regularity assumption, therefore $W(P_r, P_\theta)$ is continuous everywhere, and differentiable almost everywhere.

11. *Theorem 2.* let P be a distribution on a compact space $\mathcal{X}$ and $(P_n)_{n\in N}$ be a sequence of distributions on $\mathcal{X}$. Then, considering all limits as $n \to \infty$,
    1. The following statements are equivalent

    - $\delta(P_n, P) \to 0$ with δ the total variation distance.

- $JS(P_n, P) \to 0$ with JS the *Jensen-Shannon* divergence.

2. The following statements are equivalent
   - $W(P_n, P) \to 0$.
   - $P_n \to^{\mathcal{D}} P$ where $\to^{\mathcal{D}}$ represents convergence in distribution (weak convergence).
3. $KL(P_n \| P) \to 0$ or $KL(P \| P_n) \to 0$ imply the statement in 1.
4. The statements in 1 imply the statements in 2.

4.4.1 WGAN

1. Theorem 2 indicates that EM distance or Wasserstein-1 distance has better properties when optimized than JS divergence. However the infimum in the definition of EM distance is highly intractable. However the Kantorovich-Rubinstein duality theorm (Villani. Optimal transport: Old and New) tells us

$$W(P_r, P_\theta) = \sup_{\|f\|_L \leq 1} \{\mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]\}$$

   where the supremum is over all the 1-Lipschitz functions $f : \mathcal{X} \to R$

2. Now if we have a parameterized family of functions $\{f_w\}_{w \in \mathcal{W}}$ that are all K-lipschitz for some K, we could consider the problem

$$\sup_{w \in \mathcal{W}} \{\mathbb{E}_{x \sim P_r}[f_w(x)] - \mathbb{E}_{z \sim p(z)}[f_w(g_\theta(z))]\}$$

3. *Theorem 3.* Let P_r be any distribution. Let P_θ be the distribution of $g_\theta(Z)$ with Z a random variable with density p and g_θ a function satisfying the *regularity assumption.* Then there is a solution $f : \mathcal{X} \to R$ to the problem

$$\max_{\|f\|_L \leq 1} \{\mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]\}$$

   and we have

$$\nabla_\theta W(P_r, P_\theta) = -\mathbb{E}_{z \sim p(z)}[\nabla_\theta f(g_\theta(z))]$$

   when both terms are well defined. (They are well-defined under the condition of this theorem.)

4. *How can we find the optimal f?* One way to solve this problem is that train a neural network parameterized with weights w lying in a compact space $\mathcal{W}$ (it can make sure all the functions f_w are K-Lipschitz for some K that only depends on $\mathcal{W}$ and the critic architecture) and then backprop through $-\mathbb{E}_{z \sim p(z)}[\nabla_\theta f(g_\theta(z))]$, as we would do with a typical GAN.

   To have parameters w lie in a compact space, we could clip the weights to a fixed box (say $\mathcal{W} = [-0.01, 0.01]^l$) after each gradient update. Figures 56, 57

5. *Some analysis for the WGAN algorithm.* There are only four modifications that WGAN made on the basic GAN framework:
   1. Removing the sigmoid activation function in the last layer.
   2. Removing the log in the loss function of G/D models.
   3. Clipping the parameters so that their absolute values are smaller than a constant c.
   4. Not using any optimization methods based on momentum (like Adam method); RMSProp is the recommended method.

Figure 56: WGAN algorithm

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.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Figure 57: GAN 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 $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ 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.

6. *Benefits of WGAN* 1. Wasserstein distance is highly correlated with the generator's convergence and sample quality. Figure 58
7. *Benefits of WGAN* 2. Improved stability
   - It has no sign of mode collapse in experiments.

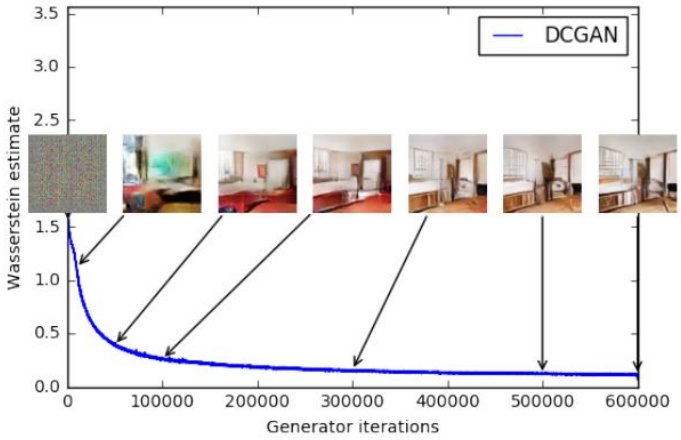


Figure 58: Wasserstein distance is highly correlated with the generator's convergence and sample quality.

- The generator can still learn when the critic perform well.
- WGANs are more robust than GANs when one varies the architectural choice for the generator.

4.4.2 Gradient penalty

1. *Difficulties of WGAN*. Weight clipping is a clearly problematic way to enforce a Lipschitz constraint. It has two difficulties:

2. Without careful tuning of the clipping threshold c, it may result in either vanishing or exploding gradients. Figure 59

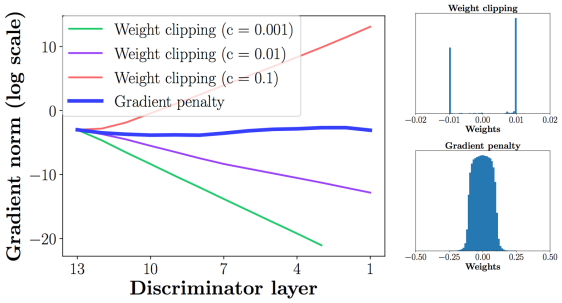


Figure 59: (left) gradient norms of deep WGAN critics during training on the Swiss Roll dateset either expolde or vanish when using weight clipping, but not when using a gradient penalty. (right) Weight clipping pushes weights towards the extremes of the clipping range, unlike gradient penalty.

3. Weight clipping reduces the capacity of the critic f and limits the capability to model complex functions. Figure 60

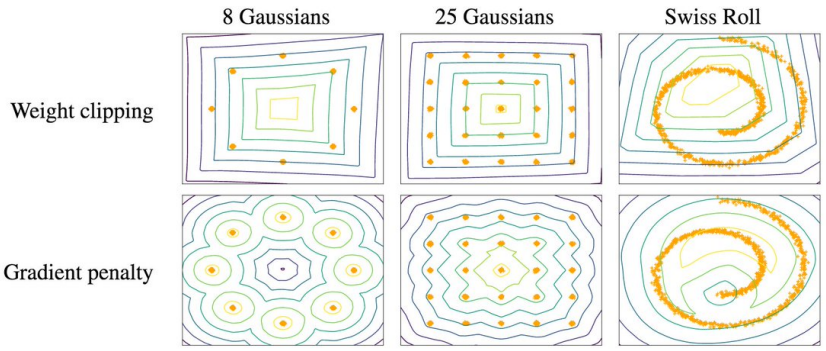


Figure 1: Value surfaces of WGAN critics trained to optimality on toy datasets. Critics trained with weight clipping fail to capture higher moments of the data distribution. The 'generator' is held fixed at the real data plus Gaussian noise.

Figure 60: Weight clipping reduces the capacity of the critic f and limits the capability to model complex functions.

4. *Proposition 1.* Let P_r and P_g be two distributions on $\mathcal{X}$, a compact metric space. Then there is a 1-Lipschitz function f^* which is the optimal solution of

$$\max_{\|f\|\leq 1} \{\mathbb{E}_{y\sim P_r}[f(y)] - \mathbb{E}_{x\sim P_g}[f(x)]\}.$$

Let π be the optimal coupling between P_r and P_g, defined as the minimizer of $W(P_r, P_g) = \inf_{\pi\in\prod(P_r,P_g)} \mathbb{E}_{(x,y)\sim\pi}[\|x-y\|]$ where $\prod(P_r, P_g)$ is the set of joint distributions $\pi(x, y)$ whose marginals are P_r and P_g, respectively. Then if f^* is differentiable, $\pi(x = y) = 0$, and $x_t = tx + (1-t)y$ with $0 \leq t \leq 1$, it holds that

$$P_{(x,y)\sim\pi}[\nabla f^*(x_t) = \frac{y - x_t}{\|y - x_t\|}] = 1.$$

5. *Corollary 1.* f^* has gradient norm 1 almost everywhere under P_r and P_g.
6. *WGAN-GP.* A differentiable function is 1-Lipschtiz if and only if it has gradients with norm at most 1 everywhere, so we consider directly constraining the gradient norm of the critics output with respect to its input. To circumvent tractability issues, we enforce a soft version of the constraint with a penalty on the gradient norm for random samples $\hat{x} \sim P_{\hat{x}}$.
7. *New objective*

$$L = E_{\tilde{x}\sim P_g}[D(\tilde{x})] - \mathbb{E}_{x\sim P_r}[D(x)] + \lambda E_{\hat{x}\sim P(\hat{x})}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]$$

8. *Algorithm.* Figure 61
9. *Some analysis.* 1. We implicitly define $P_{\hat{x}}$ sampling uniformly along straight lines between pairs of points sampled from the data distribution P_r and the generator distribution P_g. Given that enforcing the unit gradient norm constraint everywhere is intractable, enforcing it only along these straight lines seems sufficient and experimentally results in good performance.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.
while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^m \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^m -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

Figure 61: WGAN Gradient Penalty.

2. Penalty coefficient: All experiments in the paper use $\lambda = 10$, which we found to work well across a variety of architectures and datasets ranging from toy tasks to large ImageNet CNNs.
3. No critic batch normalization. Because we penalize the norm of the critics gradient with respect to each input independently, and not the entire batch.

10. *Experimental results*. WGAN-GP enhances training stability. As shown below, when the model design is less optimal, WGAN-GP can still create good results while the original GAN cost function fails. Figure 62

11. *References*.

1. M. Arjovsky and L. Bottou. Towards principled methods for training generative adversarial networks. 2017.
2. M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. arXiv preprint arXiv:1701.07875. 2017.
3. Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, Aaron Courville1, Improved Training of Wasserstein GANs. 2017

5 Sequential decision-making: from bandits to deep reinforcement learning

5.1 Intro

1. Setup
   - Question: How to learn from experience that accumulates gradually over time?
   - Specifically: How to make decisions?
2. Bandits - a clean model. Figure 63.

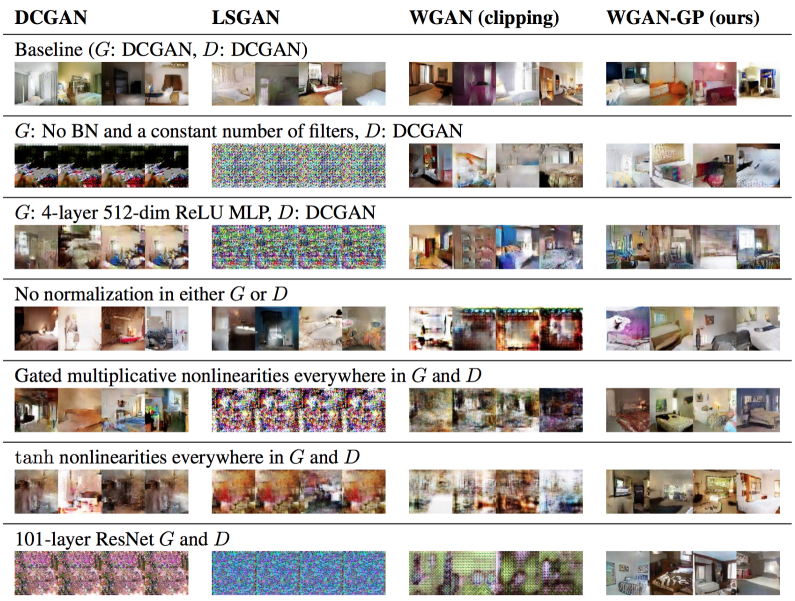


Figure 62: WGAN Gradient Penalty. Experimental results

Time	1	2	3	4	5	6	7	8	9	10	11	12
Left arm	$1	$0			$1	$1	$0					
Right arm			$1	$0								

Five rounds to go. Which arm would you play next?

Figure 63: Bandit example

3. Online advertising. Figure 64.
4. Adaptive clinical trials. Figure 65.
5. Robotics. Figure 66.
6. Examples
   - Healthcare
     - Adaptive clinical trials, e.g., oncology

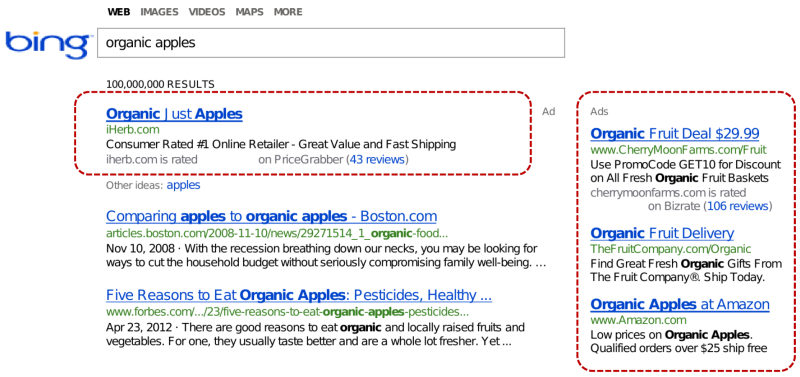


Figure 64: Online advertising

  - mHealth, e.g., smoking, stress, exercise
- Web
  - online ads, e.g., Microsoft, Google
  - dynamic pricing
  - A/B testing (different versions of website)
  - recommender systems: news, products

7. Examples
   - Robotics
     - Planning, search, control
   - Game playing
     - Classical games: Chess, Go
     - Computer games
   - ...

5.2 Bandits

1. Setup
   - *Time* $t = 1, 2, \ldots, T$
   - Need to choose an *arm* $A_t \in 1, 2, \ldots, K$,
   - Observe *reward* R_t, which depends on the arm chosen
   - Goal: maximize *returns*

$$\max \sum_{t=1}^{T} R_t$$

2. Setup (2)
   - Simplest setting: rewards are random, and distribution only depends on the arm. $R_t | A_t = k \sim P_k$ (independently of the past).
   - known as *Stochastic bandits*

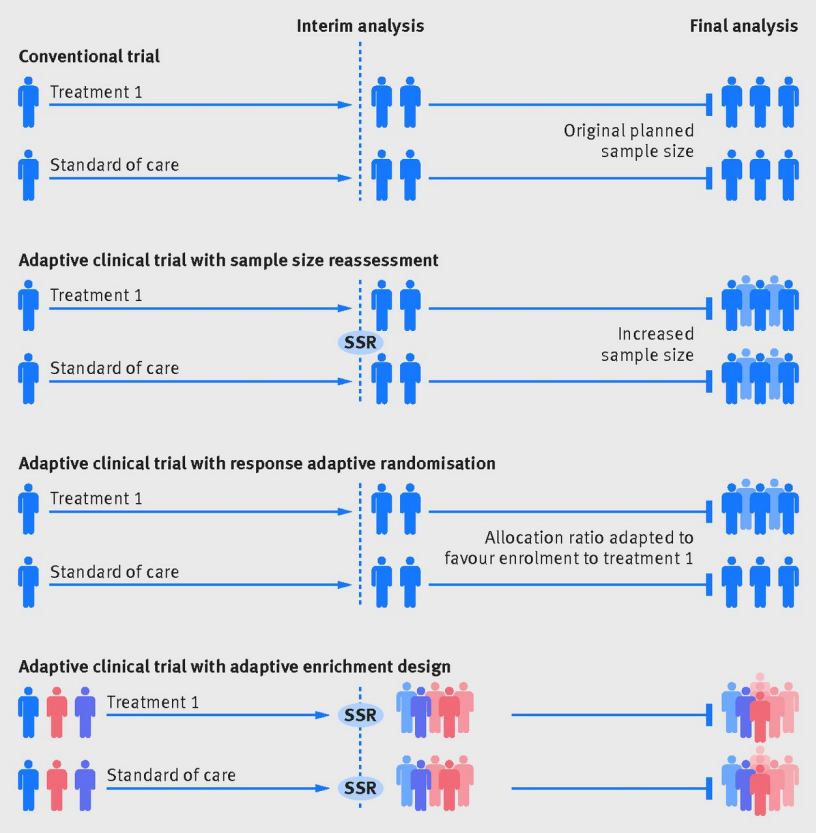


Figure 65: Adaptive clinical trials

- At each timestep, we have available the entire *history* of previous actions and rewards $A_1, R_1, A_2, R_2, \ldots, A_{t-1}, R_{t-1}$
- Our *policy* (denoted π) chooses an action, in a possibly randomized way

3. Rewards. Figure 67.
4. How to act? Any ideas?
   - Recall: we have available the entire history of previous actions and rewards $A_1, R_1, A_2, R_2, \ldots, A_{t-1}, R_{t-1}$
   - we need to choose arms in $1, \ldots, K$ to maximize sum of rewards. Rewards have a specific distribution for each arm.
5. Some possible policies
   - Initially explore all arms, then choose best (exploit) - (ETC)
   - Initially explore uniformly, then explore according to how promising the arms are - (Thompson sampling, Boltzmann/softmax, ε-greedy)
   - Estimate an upper confidence bound on the mean reward for each arm, and be greedy - UCB

Figure 66: Robotics

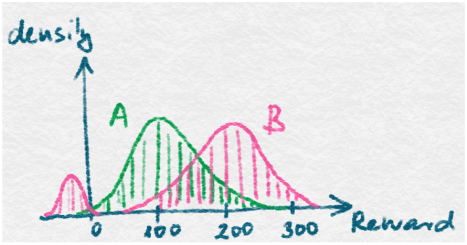


Figure 67: Reward distribution for two arms

6. Notes
   - Balance *exploration* and *exploitation*
   - Separating them is suboptimal
   - In practice the other algorithms are quite similar (if tuned)
7. Measuring success–the notion of regret
   - In the stochastic case, assume $R_t \in [0, 1]$ (or sub-Gaussian)
   - Focus only on the *mean* rewards $\mu_k = \mathbb{E}_{P_k} R_t$
   - An oracle policy would choose the best arm at each step:

     $$k^* = \arg\max_k \mu_k$$

     and $\mu_{k^*} = \mu^*$
   - A more refined comparison between policies is given by the notion of *regret*: the difference between the optimal action and the given policy:

     $$r_T = \max_k \mathbb{E}_{A_t=k} \sum_{t=1}^{T} R_t - \mathbb{E}_\pi \sum_{t=1}^{T} R_t$$

Or

$$r_T = T\mu^* - \mathbb{E}_\pi \sum_{t=1}^{T} R_t$$

8. Regret
   - Regret $r_T = T\mu^* - \mathbb{E}_\pi \sum_{t=1}^{T} R_t$
   - Minimizing regret is equivalent to maximizing return. But this is a more fine-grained measure, allows more careful comparison of policies
   - Minimize regret - important life lesson :)

5.2.1 Policies and regret analysis

1. Explore then commit (ETC)
   - ETC policy
     (a) *Explore*: Choose each arm m times
     (b) *Exploit*: Choose best arm afterwards (for $T - Km$ times)
   - **Theorem**: Suppose $K = 2$, and let $\Delta = |\mu_1 - \mu_2|$. The regret of ETC is

$$r_T \leq m\Delta + T\Delta \exp\left(-\frac{m\Delta^2}{4}\right)$$

   - Optimal $m = \lceil \frac{4}{\Delta^2} \log(\frac{T\Delta^2}{4}) \rceil$ - after this many steps, probability of error is small
   - Achieves regret

$$r_T \leq \frac{4}{\Delta}\left[\log(\frac{T\Delta^2}{4}) + 1\right] + \Delta$$

   - Worst case $\Delta \sim T^{-1/2}$, worst case regret

$$r_T = O(T^{1/2})$$

2. Notes on ETC
   - We say that ETC attains regret $T^{1/2}$
   - Proof: See tutorial by Lattimore and Szepesvari
   - Optimal up to order for $K = 2$ (but not the right constant)
   - Issues
     – Non-adaptive: need to know T, Δ (hard for $K > 2$)
     – Spend a lot of time not learning
3. Optimism Principle. Figure 68.
4. Optimism Principle - UCB Algorithm
   - Upper Confidence Bound (UCB): Estimate an upper confidence bound on the mean reward for each arm, and be greedy
   - Let $T_k(t)$ be the number of times action k was chosen until time t
   - Define the *index* (or UCB) of arm k at time t:

$$\gamma_k(t) = \hat{\mu}_k(t) + \sqrt{\frac{2\log(t^3)}{T_k(t)}}$$

   - UCB algorithm:

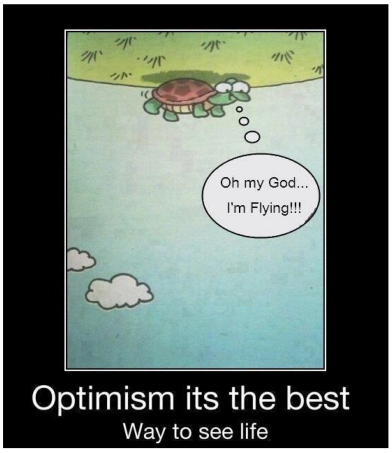


Figure 68: Optimism Principle

(a) Choose each arm once
(b) Afterwards, at time $t+1$, choose arm maximizing index $\gamma_k(t)$

5. Analyzing UCB
   - **Theorem**: Let $\Delta_i = \mu^* - \mu_i \geq 0$ be gap. The problem-dependent regret of UCB is
   $$r_T = O\left(\sum_{i:\Delta_i>0} \Delta_i + \frac{\log T}{\Delta_i}\right)$$
   - Worst case regret
   $$r_T = O(\sqrt{KT\log T})$$
6. Notes on UCB
   - May take different power $\log(t^c)$
   - May take into account variance of rewards (UCB-Tuned)
7. ε-greedy
   - ε-greedy: Let $\varepsilon > 0$ (say 0.05) be a small exploration parameter
     (a) (Explore) With probability ε choose an action uniformly at random
     (b) (Exploit) Wp $1-\varepsilon$ choose best action
   - Note: May decay $\varepsilon = \varepsilon_t \to 0$
   - Initialize $\hat{\mu}_k(0) = 0$ (or 1 - optimistic initial values)
8. Boltzmann sampling / softmax
   - Boltzmann sampling: Let $\tau > 0$ (say 0.1) be a temperature parameter
     (a) Initialize $\hat{\mu}_k(0) = 0$ (or 1)
     (b) At time $t+1$, select action k with probability proportional to
     $$P(A_t = k) \sim \exp\left[\frac{\hat{\mu}_k(t)}{\tau}\right]$$

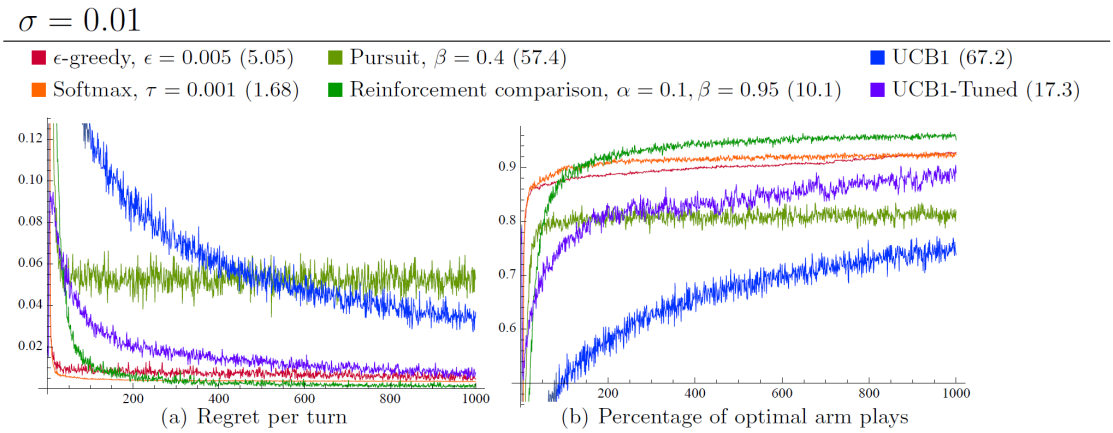
(c) Update $\hat{\mu}_k(t+1)$

9. Thompson sampling
   - Thompson sampling is a Bayesian method. Put priors on each arm's reward distribution.
     (a) At time t, select action k with probability (assuming continuous reward distributions)
     $$P(A_t = k) = P(R_k > R_j, \forall j \neq k)$$
     where $P()$ is the posterior
     (b) Update posterior of chosen action
   - Choose arm with probability that it is best
10. Thompson sampling
    - Thompson sampling for Bernoulli bandits is convenient because of conjugate priors.
    - Suppose $P_k \sim Bernoulli(\mu_k)$
    - Put prior $\mu_k \sim Beta(a, b)$ (eg 1,1)
    - If we choose action k, and sample $R_1 = 1$, the posterior becomes $\mu_k \sim Beta(a+1, b)$
    - More generally, after t steps, let S_t be the number of successes by pulling arm t, and let F_t be the number of failuers. Then the posterior is
    $$\mu_k \sim Beta(a + S_t, b + F_t)$$

5.2.2 Comparing policies

1. Which policy to use
   - Saw: UCB, ε-greedy, Boltzmann/softmax, Thompson sampling
   - Which policy to use?
   - In practice they are quite similar (if tuned)
2. Experiments
   - Experimental evaluation of several algorithms performed by Velmorel, Mohri (2005); Kuleshov, Precup (2014)
   - Protocol from Kuleshov, Precup (2014)
     - $K = 2, 5, 10, 50$, $T = 1000$ (plateau afterwards),
     - Distributions normal $P_k \sim \mathcal{N}(\mu_k, \sigma^2)$; μ_k sampled from $U[0, 1]$; $\sigma^2 = 0.01, 0.1, 1$
     - Also try other distributions P_k (only mean, variance matters)
     - Policies: UCB, UCB-Tuned, ε-greedy, Boltzmann/softmax, Pursuit, Reinforcement comparison (see book by Sutton-Barto)
     - Tune hyperparameters optimally [In practice, how??]
3. Experiments
   - Kuleshov & Precup (2014)

     The most striking observation is that the simplest algorithms, ε-greedy and Boltzmann exploration, outperform their competitors on almost all tasks. Both heuristics perform very similarly, with softmax usually being slightly better.
4. $K = 5$, $\sigma = 0.01$. Figure 69.
5. $K = 5$, $\sigma = 0.1$. Figure 70.



(a) Regret per turn (b) Percentage of optimal arm plays

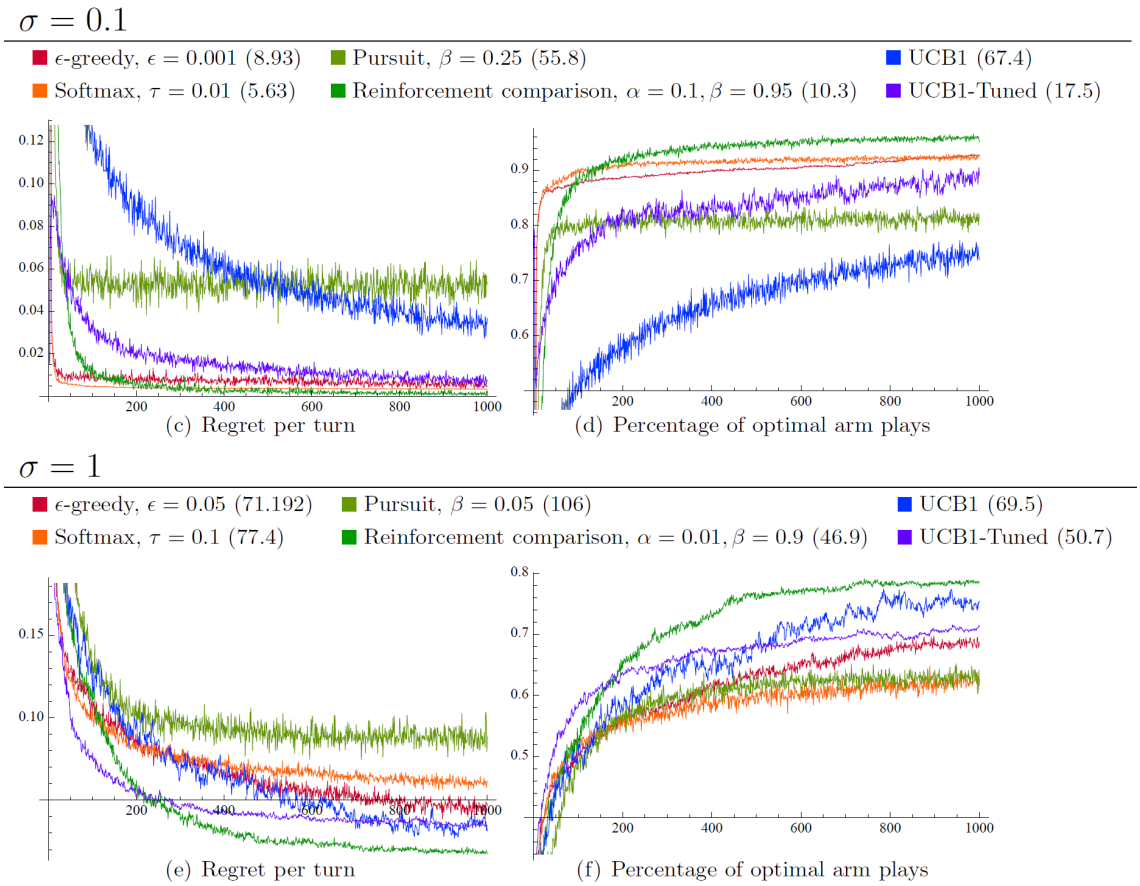
Figure 69: K=5, $\sigma = 0.01$

6. $K = 2$, $\sigma = 0.1$. Figure 71.
7. Experiments
   - Kuleshov & Precup (2014)

     In particular, softmax outperforms the other algorithms in terms of total regret on all tasks, except on the high-variance setting ($\sigma = 1$) for small and medium numbers of arms (K = 2,5,10). On these settings, softmax comes in second behind the UCB1-Tuned algorithm. In some sense, the fact that UCB1-Tuned was specifically designed to be sensitive to the variance of the arms justifies the fact that it should be superior at high values of the variance.
   - Softmax: good for low variance
   - UCB: Good for high variance
   - But not that conclusive, and how to tune?
8. Thompson sampling?
   - Empirical evaluation by Chapelle and Li (2011)
   - Thompson sampling works better than UCB
   - Protocol: K Bernoulli arms, one with $\mu_1 = 1/2$, rest with $\mu_i = 1/2 - \varepsilon$
   - Prior: $\mu_i \sim Beta(1, 1)$
   - Lower bound from Lai and Robbins (1985)

$$r_T \geq \log(T) \cdot \left[\sum_{i=1}^{K} \frac{\Delta_i}{D_{KL}(\mu_i \| \mu^*)} + o(1) \right]$$

9. Thompson sampling. Figure 72.
10. Experiments
    - Chapelle and Li (2011)



(c) Regret per turn (d) Percentage of optimal arm plays

(e) Regret per turn (f) Percentage of optimal arm plays

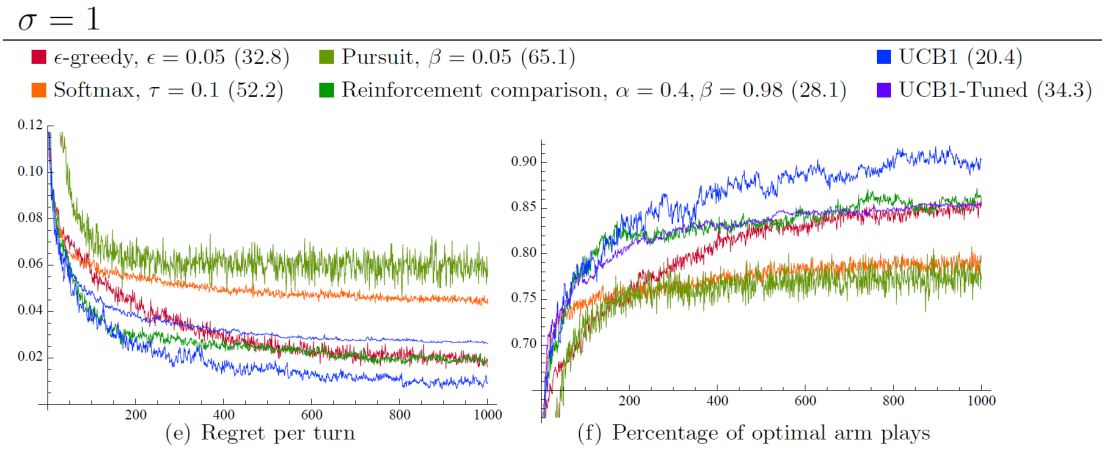
Figure 70: K=5, $\sigma = 0.1$

We can thus conclude that in these simulations, Thompson sampling is asymptotically optimal and achieves a smaller regret than the popular UCB algorithm. It is important to note that for UCB, the confidence bound is tight; we have tried some other confidence bounds, including the one originally proposed in [3], but they resulted in larger regrets.

- Theory: TS is asy optimal, attains Lai-Robbins lower bound for Bernoulli bandits and several priors (Agrawal and Goyal, 2012; Agrawal and Goyal, 2013a; Kauffmann et al., 2012).
- In some cases worst-case regret $\sqrt{KT \log(T)}$
- See tutorial by Russo, Van Roy et al, 2018

5.2.3 Adversarial bandits

1. Adversarial bandits



(e) Regret per turn (f) Percentage of optimal arm plays

Figure 1: Empirical Results for 2 arms, with different values of the variance

Figure 71: K=2, $\sigma = 0.01$

- At the beginning, adversary secretly chooses fixed rewards $R_1, R_2, \ldots, R_T$, say each in $[0,1]^K$. So $R_1(k)$ is the reward for choosing action k at the first time step
- As before: We choose actions A_t based on history, and obtain reward $R_t(A_t)$
- Hard: the adversary may choose rewards without any structure, or to "fool" us
- Regret of policy π is

$$r_T = \max_k \sum_{t=1}^{T} R_t(k) - \mathbb{E}_\pi \sum_{t=1}^{T} R_t(A_t)$$

2. Adversarial bandits - Example
   - Your friend secretly chooses rewards $x_1 \in \{0,1\}$ and $x_2 \in \{0,1\}$.
   - You implement a strategy to select $A \in \{1,2\}$ and receive reward x_A.
   - The regret is $R = \max\{x_1, x_2\} - \mathbb{E}x_A$.
3. Adversarial bandits - policies
   - Any suggestions on strategies?
   - The rewards are non-random, can be arbitrary, adversarial and may fool us
4. Adversarial bandits
   - How can we avoid being fooled?
   - If we consider a deterministic strategy, there is always an adversarial example that has maximal regret T
   - So we need to consider randomized strategies
   - A uniform random strategy already guarantees regret $T/2$. Can we do better?
   - Yes - there are policies with regret $O(\sqrt{KT})$
5. Boltzmann sampling aka Exp3 - Exponential-weights for Exploration and Exploitation

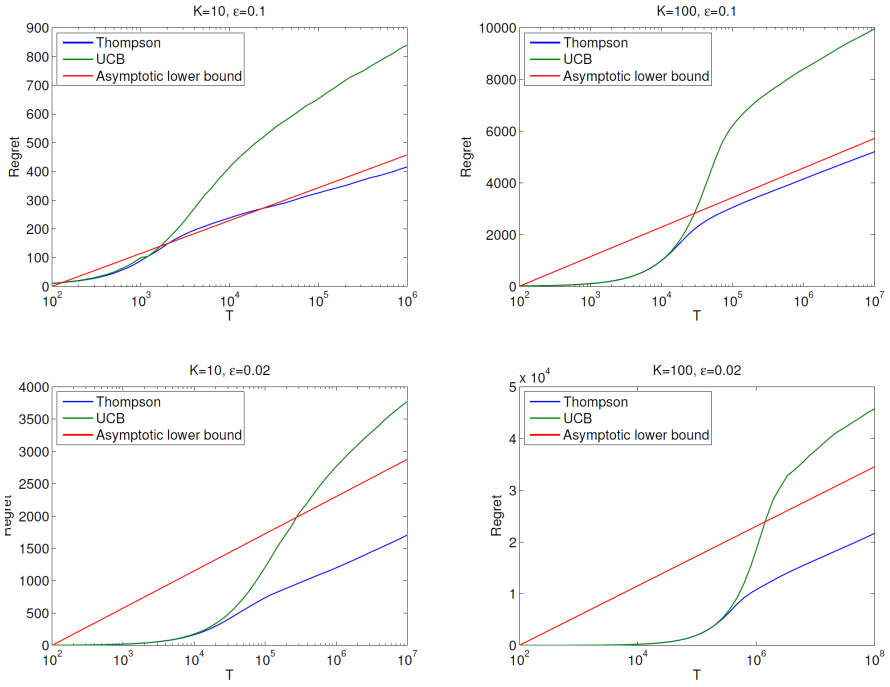


Figure 72: Thompson sampling

- Let $\tau > 0$ (say 0.1) be a temperature parameter
  (a) Initialize $\hat{\mu}_k(0) = 0$ (or 1)
  (b) At time $t+1$, select action k with probability proportional to

$$P(A_t = k) \sim \exp\left[\frac{\hat{\mu}_k(t)}{\tau}\right]$$

  (c) Update $\hat{\mu}_k(t+1) = [k\hat{\mu}_k(t) + \hat{R}_t(k)]/(k+1)$, for some unbiased estimator $\hat{R}_t$ of R_t

6. Estimators of reward
   - Key intermediate step: Find unbiased estimator $\hat{R}_t$ of R_t. Problem: we do not observe R_t
   - Denote P_t the probability distribution over actions at time t
   - *Importance weighted* estimator $\hat{R}_t$, has zeros for actions not taken, and

$$\hat{R}_t(k) = \frac{I(A_t = k)R_t(k)}{P_t(k)}$$

   - We see $\mathbb{E}[\hat{R}_t(k)|A_1, R_1, \ldots, A_{t-1}, R_{t-1}] = R_t(k)$
   - Loss-based importance-weighted estimator: preferable because takes values in $(-\infty, 1]$:

$$\hat{R}_t(k) = 1 - \frac{I(A_t = k)}{P_t(k)}(1 - R_t(k))$$

7. Regret of Exp3
   - Theorem: Let $\tau = \sqrt{K/(T \log K)}$. Then the regret of the Exp3 algorithm with the loss-based weights for adversarial bandits is
   $$r_T \leq \sqrt{2TK \log(K)}$$
8. A different perspective
   - Regret is (S^k is simplex)
   $$\begin{aligned} r_T &= \max_k \mathbb{E}\left[\sum_{t=1}^T R_t(k) - R_t(A_t)\right] \\ &= \max_{p \in S^k} \mathbb{E}\left[\sum_{t=1}^T p^\top R_t - P_t^\top R_t\right] \end{aligned}$$
   - because for a vector U, we have $\max_k U(k) = \max_{p \in S^k} p^\top U$, and we use this for $U = \sum_{t=1}^T R_t$
   - recall we denote P_t the probability distribution over actions at time t
   - *Online linear optimization on the simplex*
9. Online linear optimization on the simplex
   - Find distributions P_t sequentially, such that the following is small
   $$r_T = \max_{p \in S^k} \mathbb{E}\left[\sum_{t=1}^T (p - P_t)^\top R_t\right]$$
   - Online linear optimization on the simplex, except R_t is not observed
   - For unbiased estimator $\hat{R}_t$, we see $\mathbb{E}[\hat{R}_t(k) | A_1, R_1, \ldots, A_{t-1}, R_{t-1}] = R_t(k)$, so
   $$r_T = \max_{p \in S^k} \mathbb{E}\left[\sum_{t=1}^T (p - P_t)^\top R_t\right] = \max_{p \in S^k} \mathbb{E}\left[\sum_{t=1}^T (p - P_t)^\top \hat{R}_t\right]$$
10. Algorithm
    - *Follow the regularized leader*: for each $t = 1, \ldots,$ and for a fixed learning rate η, define P_t as
    $$P_t = \arg\max_p \eta \cdot p^\top \sum_{s=1}^{t-1} \hat{R}_s + F(p)$$
    - F is a regularizer, eg entropy $F(p) = -\sum_i p_i \log(p_i)$
    - Theorem: $r_T \leq \sqrt{2TK \log(K)}$
11. Notes
    - There is a more general picture: Online lin optimization. Allow choosing probability distributions. Two main algos: Mirror descent + Follow the regularized leader
    - Exponential weights algorithm is a special case of mirror descent (constraint set A: simplex, F negentropy)
    - In some cases the two algorithms coincide. (see Lattimore and Szepesvari for more)
12. Notes
    - Adversarial bandits are in general challenging than stochastic bandits – but in this case they are equally difficult
    - Variants: reactive adversary

5.2.4 Contextual Bandits

1. Motivation I
   - In most sequential decision making scenarios, there is some additional information available.
   - For instance, in a clinical trial, we have access to subjects' genetic and demographic information.
   - *Contextual bandit problem:* how to map user features into one of the available actions.
   - Recent applications include: mobile health, news article recommendation, etc.
2. Motivation II

   In such a decision making problem, the fundamental pattern that repeats over time is the following:

   (a) **at** a given decision point **do**
   (b) mobile phone collects tailoring variables (the context)
   (c) a decision rule maps the variables into an intervention
   (d) mobile phone records the proximal outcome (the reward)
   (e) **done**
3. Stochastic Contextual Bandits: Notation I
   (a) For simplicity, first assume two actions: $a = 0$ or $a = 1$. The data
   $$\{(X_t, R_t^0, R_t^1)\}_{t=1}^T$$
   is I.I.D. from some underlying distribution, where X_t is the context, $R_t^a, a = 0, 1$ is the reward under action $a \in \{0, 1\}$.
   (b) A *policy* or *decision* rule π maps an element X_t to an action. The *value* of a policy π is defined as:
   $$V(\pi) = E[R^{\pi(X)}]$$
   where the expectation is taken w.r.t the data-generating process of (X, R^0, R^1).
4. Stochastic Contextual Bandits: Notation II
   (a) The *expected reward function* is defined as
   $$\eta_a = E(R^a|X = x), a = 0, 1$$
   and it is easily seen that
   $$V(\pi) = E[\eta_{\pi(X)}(X)].$$
   (b) The *optimal policy* is
   $$\pi^*(x) = argmax_a\ \eta_a(x).$$
5. Stochastic Contextual Bandits: Greedy Policy
   (a) The problem can be viewed as one of estimating the expected reward functions $\eta_a(x)$. Given estimated reward $\hat{\eta}_a(x), a \in \mathcal{A}$, the greedy policy can be used:
   $$GREEDY(\hat{\eta}_a)(x) = argmax_{a\in\mathcal{A}}\ \hat{\eta}_a(x)$$
   (b) The key is to estimate the reward $\eta_a(x)$ in each arm.

6. Stochastic Contextual Bandits: Parametric Approach I

   Consider the familiar model:

   $$R_t^a = \beta_a^T X_t + \epsilon_t^a$$

   so that the expected reward is $\eta_a(x) = \beta_a^T x$. The best policy is therefore

   $$\pi^*(x) = argmax_{a \in \mathcal{A}} \ \beta_a^T x$$

   and the *regret* over T time steps is

   $$T \cdot V(\pi^*) - \sum_{t=1}^{T} E[R_t].$$

   Note $T \cdot V(\pi^*)$ is the best expected cumulative reward if you know the true β_a, while $\sum_{t=1}^{T} E[R_t]$ is the accumulated reward by the learning algorithm.

7. Stochastic Contextual Bandits: Parametric Approach II
   (a) One intuitive algorithm is as follows (assuming two arms):
      - Explore both arms for a period of time.
      - Model the reward via a regression model using the data accumulated during exploration.
      - Exploit the current estimate of the expected reward after the exploration period and takes the optimal action accordingly.

   (b) Goldenshluger and Zeevi (2013) adopts this intuition (Algorithm 1 on the next slide). They established an $O(p^3 \log T)$ regret bound.

   (c) You can also assume sparsity, extend the algorithm to high dimensional X, and improve the regret bound.

8. Stochastic Contextual Bandits: Parametric Approach III

9. SCB: Nonparametric Approach Parametric models can be restrictive. We may consider the following model instead:

   $$R_t^a = f_a(X_t) + \epsilon_t^a.$$

   Yang and Zhu (2002) combined nonparametric regression with the ϵ-greedy strategy:

10. Adversarial Contexts with Stochastic Rewards: Notation
    - The contexts are arbitrary, i.e., they are no longer I.I.D. random variables.
    - Rewards are still drawn from $\mathcal{D}^a(\cdot|x) : x \in \mathcal{X}$.
    - The optimal policy is still

      $$\pi^*(x) = argmax_{a \in \mathcal{A}} \ \eta_a(x)$$

      and the regret

      $$\sum_{t=1}^{T} \eta_{\pi^*(x_t)}(x_t) - \sum_{t=1}^{T} E[R_t].$$

    - Regret bounds need to hold uniformly over all possible sequences $\{x_t\}_{t=1}^T$ of contexts.

11. Adversarial Contexts with Stochastic Rewards: Algorithm I
    - $A^a = D_a^T D_a + I_p$, where D_a is the design matrix in arm a.
    - We have

      $$|(\hat{\beta}^a)^T x_t - \beta_a^T x_t| \leq \alpha \sqrt{x_t (A^a)^{-1} x_t}$$

      with probability at least $1 - \delta$ for any δ and x_t, with $\alpha = 1 + \sqrt{ln(2/\delta)/2}$.

Algorithm 1 Linear Response Bandit Algorithm [22]

Inputs: n_0 (initial exploration length), $\mathcal{T}_a$ (exploration times for action a), h (localization parameter to decide which estimates to use)

for $t = 1$ to $2n_0$ **do**
  Take action $A_t = 0$ or $A_t = 1$ depending on whether t is odd or even
end for
for $t = 2n_0 + 1$ to T **do**
  if $t \in \mathcal{T}_a$ **then**
    /⋆ Exploration round ⋆/
    Take action $A_t = a$
    Update $\widehat{\beta}_a$ using least squares on previous rounds when action a was taken
    Update $\widetilde{\beta}_a$ using least squares on previous exploration rounds when action a was taken
  else
    /⋆ Exploitation round ⋆/
    if $|(\widetilde{\beta}_1 - \widetilde{\beta}_0)^\top X_t| > h/2$ **then**
      Take action $A_t = \text{argmax}_a (\widetilde{\beta}^a)^\top X_t$
    else
      Take action $A_t = \text{argmax}_a (\widehat{\beta}^a)^\top X_t$
    end if
  end if
end for

Figure 73: Stochastic Contextual Bandits: Parametric Approach III

Algorithm 2 Randomized Allocation with Nonparametric Estimation [14]

Inputs: n_0 (initial exploration length), NPR (nonparametric regression procedure such as nearest neighbor regression), ε_t (sequence of exploration probabilities)

for $t = 1$ to $2n_0$ **do**
  Take action $A_t = 0$ or $A_t = 1$ depending on whether t is odd or even
end for
Get initial estimates $\widehat{f}^a$ by feeding data from previous rounds to NPR
for $t = 2n_0 + 1$ to T **do**
  Let $G_t = \text{argmax}_a \widehat{f}^a(X_t)$ // greedy action
  Let $E_t =$ action selected at random // random exploration
  With probability $(1 - \varepsilon_t)$ take action $A_t = G_t$, else $A_t = E_t$ // ε-greedy
  Collect reward R_t and feed into NPR to get updated estimate $\widehat{f}^a$ for $a = A_t$
end for

Figure 74: SYang and Zhu (2002)

Algorithm 4 LinUCB Algorithm [2]

Inputs: α (tuning parameter used in computing upper confidence bounds)
$\mathbf{A}^a = \mathbf{I}_{p\times p}, \mathbf{b}^a = \mathbf{0}_{p\times 1}$ for all a

for $t = 1$ to T **do**
  Compute $\hat{\beta}^a = (\mathbf{A}^a)^{-1}\mathbf{b}^a$ for all a // ridge regression
  Compute $U^a = (\hat{\beta}^a)^\top x_t + \alpha\sqrt{x_t^\top (\mathbf{A}^a)^{-1} x_t}$ for all a // upper confidence bound
  Take action $A_t = \text{argmax}_a U^a$ and observe reward R_t
  For $a = A_t$, update $\mathbf{A}^a = \mathbf{A}^a + x_t x_t^\top$, $\mathbf{b}^a = \mathbf{b}^a + R_t x_t$
end for

Figure 75: Adversarial Contexts with Stochastic Rewards 2

Algorithm 5 Thompson Sampling Algorithm [2]

Inputs: σ^2 (variance parameter used in the prior and in the reward linear model)
$\mathbf{A}^a = \mathbf{I}_{p\times p}, \mathbf{b}^a = \mathbf{0}_{p\times 1}$ for all a

for $t = 1$ to T **do**
  Compute $\hat{\beta}^a = (\mathbf{A}^a)^{-1}\mathbf{b}^a$ for all a
  Sample $\widetilde{\beta}^a$ from $\text{NORMAL}(\hat{\beta}^a, \sigma^2(\mathbf{A}^a)^{-1})$ for all a // Sample from the posterior
  Take action $A_t = \text{argmax}_a (\widetilde{\beta}^a)^\top x_t$ and observe reward R_t
  For $a = A_t$, update $\mathbf{A}^a = \mathbf{A}^a + x_t x_t^\top$, $\mathbf{b}^a = \mathbf{b}^a + R_t x_t$
end for

Figure 76: Adversarial Contexts with Stochastic Rewards 2

12. Adversarial Contexts with Stochastic Rewards: Algorithm II A Bayesian perspective involves putting a prior on the parameters β^a and draw samples from posterior of β^a.
13. Fully Adversarial Contextual Bandits I
    - One protocol is as follows:
      (a) nature generates $\{(x_t, \mathcal{D}_t^0(\cdot|x), \mathcal{D}_t^1(\cdot|x))\}_{t=1}^T$ in advance
      (b) **for** t = 1 to T **do**
      (c) receive context x_t
      (d) algorithm takes action A_t
      (e) receive reward R_t from $\mathcal{D}_t^{A_t}(\cdot|x)$ with expectation $\eta_t^{A_t}(x_t)$
      (f) **end**
    - Slivkins (2014) gave an algorithm with regret bound of $O(T^{1-1/(2+d_\mathcal{X})}(\log T))$ where $d_\mathcal{X}$ is the covering dimension of $\mathcal{X}$, which satisfies $d_\mathcal{X} \leq p$ when $\mathcal{X} \subseteq \mathbb{R}^p$.
14. Fully Adversarial Contextual Bandits II
    - Another protocol is as follows:
      (a) nature generates $\{(x_t, r_t^0, r_t^1)\}$ in advance
      (b) **for** t = 1 to T **do**
      (c) receive context x_t
      (d) algorithm takes action A_t
      (e) receive reward $R_t = r_t^{A_t}$

(f) **end**

- Regret is now defined as

$$\max_{\pi \in \Pi} \sum_{t=1}^{T} r_t^{\pi(x_t)} - \sum_{t=1}^{T} E[R_t]$$

where Π is a fixed class of policies.

- When $x_t = x$, this reduces to the multi-armed bandit problem with K arms and Exp family algorithms work.

5.2.5 Software and Miscellanea

1. Software
   - Not that widely implemented
   - Links to a few software packages (in Python) have been provided in the syllabus
   - Experimental evaluations on real data tricky - how do we find the reward for the actions not taken?
2. Terminology
   - Sequential experimental design
   - Online learning
   - Active/Interactive learning
   - Reinforcement learning

5.3 Reinforcement learning

5.3.1 Setup

1. One of the best introductions: http://karpathy.github.io/2016/05/31/rl/
2. Algorithm of choice: Policy gradient

   Policy Gradients: Run a policy for a while. See what actions led to high rewards. Increase their probability.
3. Good post on limitations of AlphaGo:

   https://medium.com/@karpathy/alphago-in-context-c47718cb95a5
4. Notes below adapted from David Silver's Lecture Notes

5.3.2 L1. Introduction

1. Characteristics of RL. What makes reinforcement learning different from other machine learning paradigms?
   - There is no supervisor, only a *reward* signal
   - Feedback is delayed, not instantaneous
   - Time really matters (sequential, non i.i.d data)
   - Agents actions affect the subsequent data it receives
2. Examples of RL. Figure 77
   - Fly stunt manoeuvres in a helicopter
   - Defeat the world champion at Backgammon

Figure 77: Example RL

- Manage an investment portfolio
- Control a power station
- Make a humanoid robot walk
- Play many different Atari games better than humans

3. The RL Problem. Reward:
   - A reward R_t is a scalar feedback signal
   - Indicates how well agent is doing at step t
   - The agent's job is to maximize cumulative reward

   Reinforcement learning is based on the **reward hypothesis**.

   Definition 5.1 (Reward Hypothesis)**.** *All goals can be described by the maximisation of expected cumulative reward*

4. Sequential Decision Making
   - Goal: select actions to maximise total future reward
   - Actions may have long term consequences
   - Reward may be delayed
   - It may be better to sacrifice immediate reward to gain more long-term reward
5. Agent and Environment. Figure 78
   - At each step t the agent:
     - Executes action A_t
     - Receives observation O_t
     - Receives scalar reward R_t
   - The environment:
     - Receives action A_t
     - Emits observation O_{t+1}
     - Emits scalar reward R_{t+1}
   - t increments at env. step

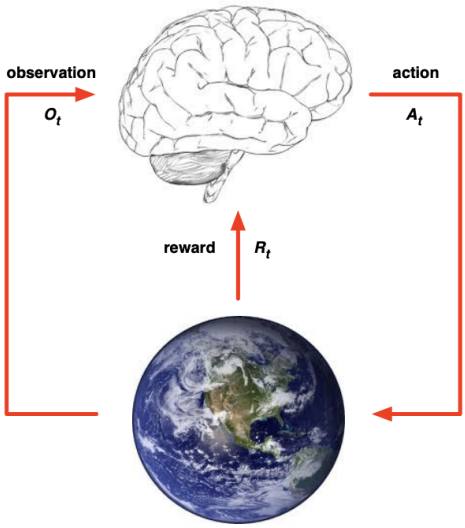


Figure 78: Agent and Environment

5.3.3 Definitions

1. History and State.

Definition 5.2 (History). *The **history** is the sequence of observations, actions, rewards*

$$H_t = O_1, R_1, A_1, \ldots, A_{t-1}, O_t, R_t.$$

In other words, all observable variables up to time t.

Definition 5.3 (State). *State is the information used to determine what happens next. Formally, it is a function of the history*

$$S_t = f(H_t).$$

Definition 5.4 (Environment State). *The environment state S_t^e is the environment's private representation*

- *i.e. whatever data the environment uses to pick the next observation/reward.*
- *The environment state is not usually visible to the agent. Even if S_t^e is visible, it may contain irrelevant information.*

Definition 5.5 (Agent State). *The agent state S_t^a is the agent's internal representation.*

- *i.e. whatever information the agent uses to pick the next action*
- *i.e. it is the information used by reinforcement learning algorithms*
- *It can be any function of history: $S_t^a = f(H_t)$*

Definition 5.6 (Information State). *An information state (a.k.a. Markov state) contains all useful information from the history.*

Definition 5.7 (Markov State). *A state S_t is Markov if and only if*

$$P[S_{t+1} \mid S_t] = P[S_{t+1}|S_1, ..., S_t].$$

In other words, "The future is independent of the past given the present," or once the state is known, the history may be thrown away.

(The environment state and the whole history are both Markov)

2. Environment

   Definition 5.8 (Full Observability)**.** *Agent directly observes environment state*

   $$O_t = S_t^a = S_t^e.$$

   - *Agent state = environment state = information state*
   - *Formally, this is a* ***Markov decision process (MDP)****. (Next)*

   Definition 5.9 (Partial Observability)**.** *Agent indirectly observes environment. (ex: a robot with camera vision isn't told its absolute location, a poker playing agent only observes public cards)*

   - *Now, agent state $\neq$ environment state.*
   - *Formally, this is a* ***partially observable Markov decision process (POMDP)****.*
   - *Agent must construct its own state representation S_t^a:*
     - *Complete history: $S_t^a = H_t$*
     - ***Beliefs*** *of environment state: $S_t^a = (P[S_t^e = s^1], \ldots, P[S_t^e = s^n])$*
     - *Recurrent neural network: $S_t^a = \sigma\left(S_{t-1}^a W_s + O_t W_o\right)$*

3. RL Agent.

   An RL agent may include one or more of these components:

   - Policy: agent's behaviour function
   - Value function: how good is each state and/or action
   - Model: agent's representation of the environment

4. Policy.

   A policy is the agent's behaviour. It is a map from state to action, e.g.

   - Deterministic policy: $\pi(s) = a$
   - Stochastic policy: $\pi(a \mid s) = P[A_t = a | S_t = s]$

5. Value.

   Value function is a prediction of future reward. Used to evaluate the goodness/badness of states and therefore to select between actions, e.g

   $$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} \cdots \mid S_t = s]$$

6. Model.

   A model predicts what the environment will do next

   - Transitions: $\mathcal{P}$ predicts the next state
   - Rewards: $\mathcal{R}$ predicts the next (immediate) reward

   $$\mathcal{P}_{ss'}^a = P[S_{t+1} = s' \mid S_t = s, A_t = a]$$
   $$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

7. Categorizing RL Agents

   - Value Based

  - No Policy (Implicit)
  - Value Function
- Policy Based
  - Policy
  - No Value Function
- Actor Critic
  - Policy
  - Value Function
- Model Free
  - Policy and/or Value Function
  - No Model
- Model Based
  - Policy and/or Value Function
  - Model

8. Fundamental distinction in RL: Model free vs Model based.
9. Problems within RL:
   sequential decision making (RL/learn the environment); planning (known environment)
   Exploration/Exploitation
   Prediction and Control

5.3.4 Relation to Contextual Bandits

1. Review Contextual Bandits. Re-introducing the contextual bandit problem using notation borrowed from RL.

   for $t = 1$ to T:
   Learner sees context $S_t \in \mathcal{S}$
   Learner selects action $A_t \in \mathcal{A}$ (with consideration to S_t)
   Learner receives reward $R_t = R_t^{A_t}$
   end for

   The optimal policy is one which maximizes the value function for every context $s \in \mathcal{S}$: $\pi^*(s) = \arg\max_a \mathbb{E}[R_t^a \mid S_t = s]$.

   Multi-armed bandit problems are thought of as reinforcement learning problems with single state.

2. Reinforcement Learning. Now, there are T episodes as before, but each episode can have a series of state-action pairs.

   for $t = 1$ to T:
   Learner sees $S_{t,0} \in \mathcal{S}$
   for $k = 0$ to $K - 1$:
   Learner selects action $A_{t,k} \in \mathcal{A}$
   Learner sees state $S_{t,k+1}^{A_{t,k}} \in \mathcal{S}$
   Learner receives reward $R_{t,k} = R(S_{t,k}^{A_{t,k-1}}, A_{t,k}, S_{t,k+1}^{A_{t,k}})$
   end for

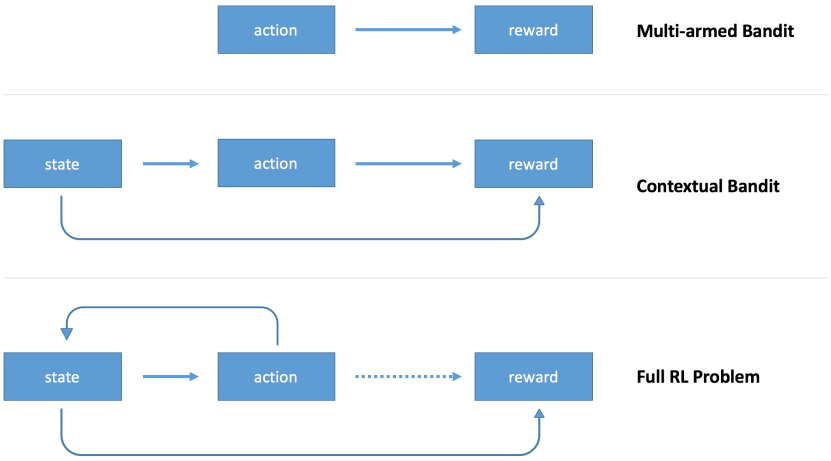


Figure 79: Comparing bandits and RL

end for

Policy is now a vector $\pi = (\pi_0, \ldots, \pi_{K-1})$ so that $A_{t,0} = \pi_0(S_{t,0}), A_{t,1}\pi_1(S_{t,1}^{A_{t,0}}), \ldots, A_{t,K-1} = \pi_{K-1}(S_{t,K-1}^{A_{t,K-2}})$.

3. Comparing bandits and RL. Figure 79
   - Bandit problems are thought of being a special case of reinforcement learning.
   - It is harder to get regret bound results for general RL problems.
   - Both problems involve maximizing some cumulative reward.
   - Both involve aspects of balancing *exploration* and *exploitation.*

5.3.5 Markov Decision Process

1. MDP. **Markov decision processes** formally describe an environment for reinforcement learning where the environment is fully observable.
   - i.e. The current state completely characterizes the process
   - Almost all RL problems can be formalised as MDPs
     - Partially observable problems can be converted into MDPs
     - Bandits are MDPs with one state

   Remember the **Markov property**: a state S_t is Markov if and only if $P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t]$.

   For a Markov state s and successor state s', the *state transition probability* is defined by

$$\mathcal{P}_{ss'}^{a} = P[S_{t+1} = s' \mid S_t = s].$$

   A state transition matrix defines transition probabilities from all states s to successor state s'. (Let $|\mathcal{S}| = n$.)

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

2. Markov Chain Review

   Definition 5.10 (Markov Chain)**.** *A Markov chain (or Markov process) is a tuple* $< \mathcal{S}, \mathcal{P} >$

   - $\mathcal{S}$ *is a finite set of states*
   - $\mathcal{P}$ *is a transition probability matrix.*

   Definition 5.11 (State Probability Vector)**.** *A vector* $q_t = (q_1^t, \ldots, q_{|\mathcal{S}|}^t)$*, where* q_s^t *means that the Markov chain is in state* s *at time* t*. Note* $q^{t+1} = q^t\mathcal{P}$*.*

   State probability vector such that $q\mathcal{P} = q$ *is called* ***stationary distribution****.*

   (Others: irreducible, aperiodic, ergodic, FTMC)

3. MRP and MDP.

   Definition 5.12 (Markov Reward Process)**.** *A Markov reward process is a Markov chain with values. It is formally specified by a 4-tuple* $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$*.*

   Reward:

   $$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$$

   Return

   $$G_t = \sum_{k=0}^{\infty} R_{t+k+1}\gamma^k$$

   Value function

   $$v(s) = \mathbb{E}[G_t | S_t = s]$$

4. Bellman equation for MRPs

   $$v(s) = \mathcal{R}_s + \gamma \sum_{s'} P_{ss'} v(s')$$

   Vector form

   $$v = \mathcal{R} + \gamma P v$$

   Value of present is current reward plus discounted future reward.
   Can solve it in closed form. [Why do iteration then?].

5. MDP:

   Definition 5.13 (Markov Decision Process)**.** *A Markov decision process is a Markov reward process with decisions (actions). It is formally specified by a 5-tuple* $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$*. (Probabilities and rewards can be action-dependent.)*

   Reward:

   $$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

   Policy:

   $$\pi(a \mid s) = P[A_t = a | S_t = s]$$

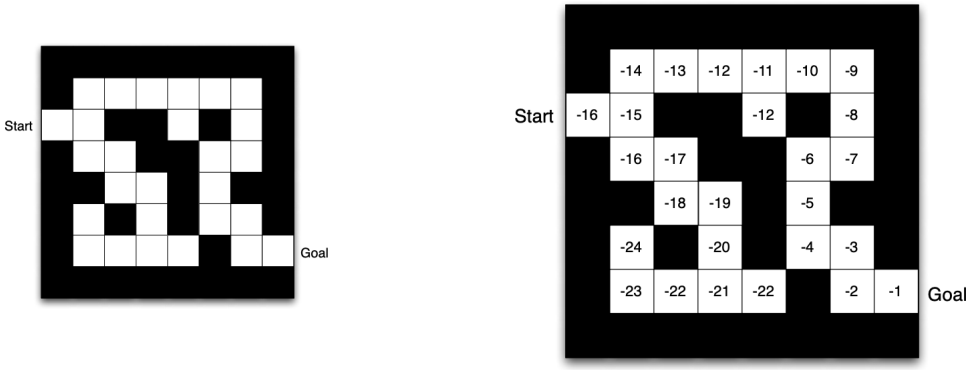


Figure 80: Maze and Value-based

Action-Value function (Q-function)

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Bellman expectation equations:
V-Q

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$$

Q-Q

$$\begin{aligned} q_\pi(s,a) &= R_s^a + \gamma \sum_{a',s'} q_\pi(s',a') P[S_{t+1} = s', A_{t+1} = a' | S_t = s, A_t = a] \\ &= R_s^a + \gamma \sum_{a',s'} q_\pi(s',a') \pi(a'|s') P_{ss'}^a \end{aligned}$$

Q-V

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'} v_\pi(s') P_{ss'}^a$$

V-V

$$v_\pi(s) = \sum_a \pi(a|s)[R_s^a + \gamma \sum_{s'} v_\pi(s') P_{ss'}^a]$$

6. Maze Example. Value Based. Figure 80. Policy Based. Model Based. Figure 81.
   - Rewards: -1 per time-step
   - Actions: N, E, S, W
   - States: Agents location

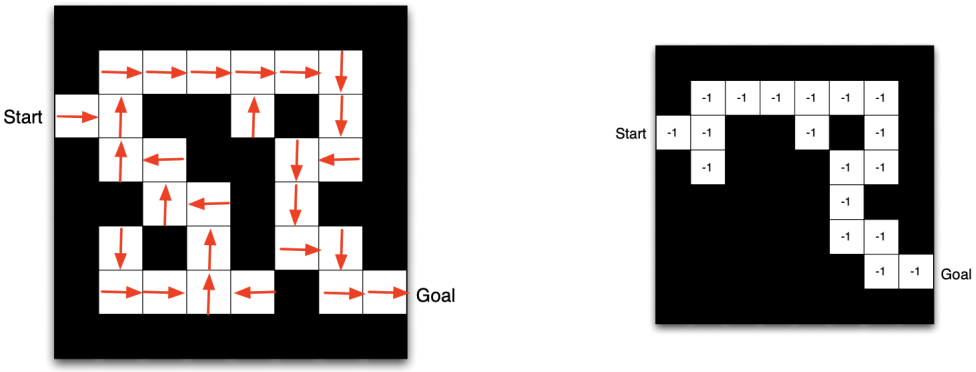


Figure 81: Policy and Model-based

7. Optimal state-value function

$$v_*(s) = \max_\pi v_\pi(s)$$

Optimal AV-function

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

This solves the MDP. For any given state, just choose the action that maximizes it.

Partial order on policies: $\pi \geq \pi'$ if the value function is always larger.

Theorem: For an MDP, there is an optimal policy. All optimal policies achivele optimal value and AV function.

8. Bellman optimality equations

V-Q

$$v_*(s) = \max_a q_*(s,a)$$

Q-V

$$q_*(s,a) = R_s^a + \gamma \sum_{s'} v_*(s') P_{ss'}^a$$

V-V

$$v_*(s) = \max_a [R_s^a + \gamma \sum_{s'} v_*(s') P_{ss'}^a]$$

9. Iterative solution methods: Value/Policy iteration, Q-learning

5.3.6 Planning by dynamic programming

1. Idea: turn fixed point equations into algorithms.
2. Recall Bellman expectation equation for value:

$$v = \mathcal{R} + \gamma P v$$

   Policy evaluation: Instead, use

$$v^{k+1} = \mathcal{R} + \gamma P v^k$$

3. Policy iteration: Alternate (1) evaluation (2) greedy-type improvement
4. Policy improvement theorem: "Policy iteration solves MDP"
5. Alternatives: approximate evaluation. Value iteration: just one iteration of Bellman optimality backup

5.3.7 Model-free prediction

1. There is an MDP but we don't know it. Idea: take random samples and estimate its value
2. MC (V-)learning: Goal. Learn v_π from episodes of experience under policy π.
   Rollouts: data $S_1, A_1, R_1, \ldots \sim \pi$
   Recall $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$. Estimate it by MC
3. First visit: first time state s is visited, increment counter $N(s) \leftarrow N(s) + 1$. $S(s) \leftarrow S(s) + G_t$ (needs entire path)
   $\hat{V}(s) = S(s)/N(s)$
   By LLN $\hat{V}(s) \to v_\pi(s)$ as $N(s) \to \infty$.
4. Every-visit: so same
   Question: which one is better?
5. TD-Learning: Replace G_t by $R_{t+1} + \gamma V(S_{t+1})$. [It is referred to as an "estimate" or "bootstrapping"]
   Formally, algorithm update is

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

   [This feels very inefficient and biased at the beginning. How can it work?]
   Can learn from incomplete samples
   Theorem: Tsitsiklis and Van Roy. Linear TD(0) converges close to optimum.
6. What happens with finite data and infinite iterations?
   MC $\to$ minimum MSE
   TD(0) $\to$ ML Markov Model
7. Unified view of RL. Figure 82
8. How to estimate the return? There are alternatives. $TD(\lambda)$ (may not converge)
   Start with n step lookahead

$$G_t^n = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

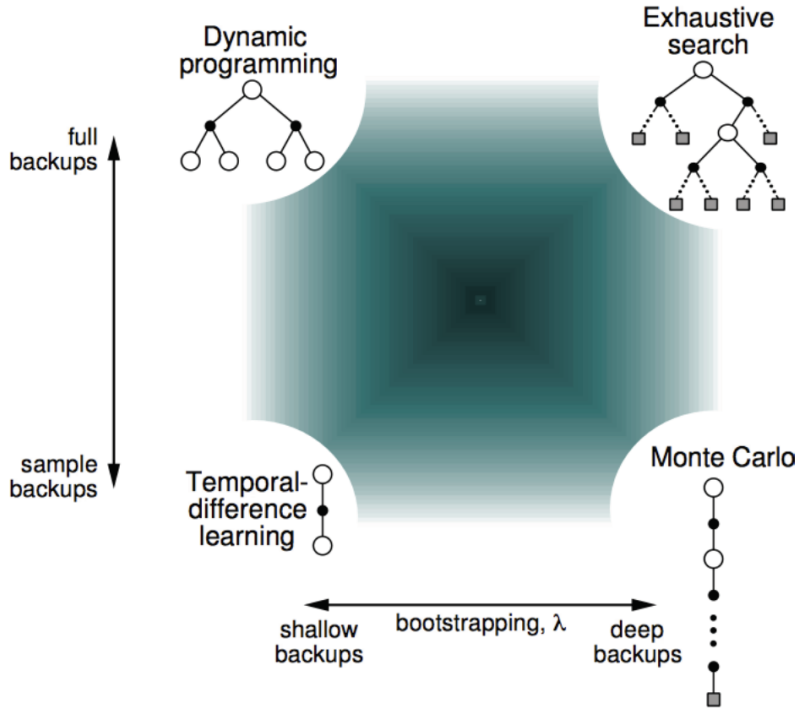


Figure 82: Unified view of RL

Construct λ return (Forward view)

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^n$$

Backward view gives an efficient implenentation in terms of eligibility traces

$$E_t(s) = \gamma\lambda E_{t-1}(s) + I(S_t = s)$$

Update

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(S) \leftarrow V(S) + \alpha\delta_t E_t(s)$$

BW and FW are equivalent for $\lambda = 1$, but not more generally [?]

5.3.8 Model-free control

1. On-policy learning:
   Evaluation: MC Policy evaluation of q_π (ie use Model-free prediction)
   Improvement: ε-greedy

2. Theorem: For any ε-greedy policy π, the ε-greedy policy π' wrt q_π is an improvement: $v_{\pi'}(s) \geq v_\pi(s)$
3. GLIE: Greedy in the Limit with Infinite Exploration
   Set $\varepsilon := \varepsilon_k \to 0$, e.g., $1/k$
   Theorem: GLIE-MC control converges to optimal Q-function
4. TD? "SARSA". Work with Q

$$\delta Q(S, A) \leftarrow \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

   Similar theory, use Robbins-Monro step sizes. Similar TD.
5. Off-policy learning. Target policy π, behavior policy μ.

$$\delta Q(S_t, A_t) \leftarrow \alpha[R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)]$$

   The trick is to sample A' from the target policy. So we follow μ, but we evaluate Q with respect to π
6. Special case/Important application: Off policy control.
   Here the target policy is greedy with respect to $Q(s, a)$ (of the current behavior policy). The behavior policy is ε-greedy with respect to $Q(s, a)$ (that's how you choose s')
   "SARSAMAX"

$$\delta Q(S, A) \leftarrow \alpha[R + \gamma \max_{a'} Q(S', a') - Q(S, A)]$$

5.3.9 Scaling up RL. Value function approximation

1. Approximate $\hat{v}(s, w) \approx v_\pi(s)$
   How to do it? Nees to set up an objective, compute the gradient, and approximate it
   Goal:

$$\min_w J(w) := \mathbb{E}_\pi (v_\pi(s) - \hat{v}(s, w))^2$$

   GD

$$\delta w = -1/2 \cdot \alpha \nabla_w J(w) = \alpha \mathbb{E}_\pi (v_\pi(s) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)$$

   SGD

$$\delta w = \alpha (v_\pi(s) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w)$$

2. Example: Features $x(s)$. Linear approximation: $\hat{v}(s, w) = x(s)^\top w$. In that case $\nabla_w \hat{v}(s, w) = x(s)$ so

$$\delta w = \alpha (v_\pi(s) - \hat{v}(s, w)) \nabla_w \hat{v}(s, w) x(s)$$

3. Incremental prediction: approximate v_π in the usual way, by MC, TD (may not converge)
   Effectively, S_i, G_i become the (non-independent) training data
4. Q-function approximation: similar
   Use in control: similar

5. Batch methods: seek to find the best fitting value function. experience replay. Just like I said above, Effectively, S_i, G_i become the (non-independent) training data
6. Deep Q-Networks (DQN) uses experience replay and fixed Q-targets. FQT fixes Q function for a while, and optimizes another one for improved convergence.

   Was used in Nature paper V Mnih - 2015 "Human-level control through deep reinforcement learning"

5.3.10 Policy gradient

1. Instead of $q(a, s)$, work with $\pi(a|s)$, or approximate by $\pi(a|s, \theta)$
2. Objective function:

   Episodic environment - Start value

   $$J_1(\theta) = v^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}(v_1)$$

   Continuing environment - average value

   $$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) v^{\pi_\theta}(s_1)$$

   Here d^{π_θ} is the strationary distribution of the MC for π_θ

   Average per time [?]

   $$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) R_s^a$$

3. Policy Gradient: $\Delta\theta = \alpha \nabla_\theta J(\theta)$

   Policy gradient theorem: For any objective $J \in J_1, J_{avR}, J_{avV}/(1-\gamma)$, we have

   $$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)$$

   Provides unbiased stochastic gradient $\nabla_\theta \log \pi_\theta(s_t, a_t) v_t$

   Algorithm: MC Policy Gradient (aka REINFORCE). Tends to be sample inefficient.
4. Actor-Critic methods. Combine value function approximation and policy gradient.
5. Variance reduction methods. Subtract any baseline function $B(s)$ from Q without changing expectation

   $$\mathbb{E}_{\pi_\theta} \nabla_\theta \log \pi_\theta(s, a) B(s) = \sum_s d^{\pi_\theta}(s) B(s) \nabla_\theta \sum_a \pi_\theta(s, a) = 0$$

   Ideal baseline $V^{\pi_\theta}(s)$.

   Advantage function $A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)$

   Estimate: [Something interesting] The average of the DS error is the advantage function

   $$\mathbb{E}_{\pi_\theta}[r + \gamma v^{\pi_\theta}(s') - v^{\pi_\theta}(s) | s, a] = Q^{\pi_\theta}(s, a) - v^{\pi_\theta}(s)$$

   So just need one value function v, and can still achieve variance reduction

5.3.11 Integrating learning and planning

1. Learn model from experience. Use planning to construct value function or policy.

   [From a statistical point of view this seems inefficient. Why not directly construct value function/policy from experience?]

2. Model based RL: Model $S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$

   $R_{t+1} \sim R_\eta(R_{t+1}|S_t, A_t)$

   Fit models for both. Transform experience $S_1, A_1, R_1, \ldots, S_T$ into supervised learning datapoints

   [Why, when? When learning a value function is somehow hard. When the system is more easily described by rules.]

3. Planning with a model: Sample-based planning. Use model to generate samples. Use model-free RL for prediction and control (ie how to act)

4. Integrated architectures: "Dyna", exactly combine real and simulated data

5. Simulation based search: Pushing planning further

   Forward search: solve MDP starting from now. So effectively this localizes the problem to start from the present search. (need only value functions of states that you can reach)

   Simulation based search: simulate MDP starting from now (using model). Use model-free RL for control

   e.g., MCTS, Monte Carlo Tree Search. = MC control for simulated experience, starting from root state. [simulate experience according to model of state, reward. act according to current policy. use MC to evaluate value function of policy. improve by ep-greedy]

   (a) Given a model M

   (b) Simulate K episodes from current state s_t using current simulation policy π

   $$\{s_t, A_t^k, R_{t+1}^k, S_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M, \pi$$

   (c) Build search tree containing visited states and actions

   (d) Evaluate states $Q(s, a)$ by mean return of episodes from s, a

   $$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^{K} \sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u \to_P q_\pi(s, a)$$

   (e) After search is finished, select current (real) action with maximum value in search tree

   $$a_t = \arg\max_a Q(s_t, a)$$

   In MCTS, the simulation policy π improves

   Each simulation consists of two phases (in-tree, out-of-tree)

   (a) Tree policy (improves): pick actions to maximise $Q(S, A)$

   (b) Default policy (fixed): pick actions randomly

6. Repeat (each simulation)

   (a) Evaluate states $Q(S, A)$ by Monte-Carlo evaluation

   (b) Improve tree policy, e.g. by ε greedy(Q)

   Monte-Carlo control applied to simulated experience. Converges on the optimal search tree, $Q(S, A) \to q_*(S, A)$

8. AlphaGo paper: uses MCTS;
   (a) Highly selective best-first search
   (b) Evaluates states dynamically (unlike e.g. DP)
   (c) Uses sampling to break curse of dimensionality
   (d) Works for black-box models (only requires samples)
   (e) Computationally efficient, anytime, parallelisable

   Can also use: TD

5.3.12 Hierarchical RL

1. Problem & Motivation Weaknesses of RL
   Sample inefficiency
   Sparse reward environments
   Large state, action space environments
   Unintuitive
   Generalization and abstraction
   Hold on ... we solved Go!
2. Hierarchical Reinforcement Learning = Reinforcement Learning + Temporal Abstraction
   Decompose goal into subtasks
   Learn low-level policies to solve small subtasks Compose low-level policies into longer-term, more abstract strategies to achieve goal
   (a) Denser rewards
   (b) Transfer learning via subproblem re-use
   (c) Distill state/action space into cohesive subspaces
3. Feudal Learning
   Markov Options
   HAMs
   MAXQ
4. Feudal Learning Introduction (Dayan & Hinton, 93)
   Managers - Submanagers - Workers
   (a) Reward hiding:
   A submanager receives reward if and only if it achieves the goal set for it by its manager
   No reward if manager goal attained but not submanager goal attained
   Receives reward if goal attained but manager goal not attained
   (b) Information hiding:
   Information hidden downwards submanagers do not know their managers task
   Information hidden upwards managers do not know how their sub-managers have assigned workers to complete task
   (c) Maze Task, Figure 83
   To illustrate this feudal system, consider a standard maze task (Barto, Sutton & Watkins, 1989) in which the agent has to learn to find an initially unknown goal. The grid is split up at successively finer grains (see figure 1) and managers are assigned to separable parts of the maze at each level. So, for instance, the level 1 manager of area 1-(1,1) sets the tasks for and reinforcement given to the level 2 managers for

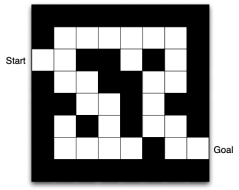


Figure 83: Maze Task.

areas 2-(1,1), 2-(1,2), 2-(2,1) and 2-(2,2). The successive separation into quarters is fairly arbitrary however if the regions at high levels did not cover contiguous areas at lower levels, then the system would not perform very well.
This shows how the maze is divided up at different levels in the hierarchy. The U shape is the barrier, and the shaded square is the goal. Each high level state is divided into four low level ones at every step
At all times, the agent is effectively performing an action at every level. There are five actions, NSEW and *, available to the managers at all levels other than the first and last. NSEW represent the standard geographical moves and * is a special action that non-hierarchical systems do not require. It specifies that lower level managers should search for the goal within the confines of the current larger state instead of trying to move to another region of the space at the same level. At the top level, the only possible action is *; at the lowest level, only the geographical moves are allowed, since the agent cannot search at a finer granularity than it can move.
N, S, E, W: Move to region in given cardinal direction at current level
*: pass control to sub-managers to search for goal within current region at finer grain
A_1 : {*}
$A_{1-(n-1)}$: {N, S, E, W, *}
A_n : {N, S, E, W}
State:
Action selected by manager above
Location of agent on the board in the granularity below
Tabular Q-values updated at all levels where a transition occurred, if the transition was ordered at all lower levels
Each manager maintains Q values (Watkins, 1989; Barto, Bradtke & Singh, 1992) over the actions it instructs its sub-managers to perform, based on the location of the agent at the subordinate level of detail and the command it has received from above. So, for instance, if the agent currently occupies 3-(6,6), and the instruction from the level 0 manager is to move South, then the 1-(2,2) manager decides upon an action based on the Q values for NSEW giving the total length of the path to either 2-(3,2) or 2-(4,2). The action the 1-(2,2) manager chooses is communicated one level down the hierarchy and becomes part of the state determining the level 2 Q values.
When the agent starts, actions at successively lower levels are selected using the standard Q -learning softmax method and the agent moves according to the finest grain action (at level 3 here). The Q values at every level at which this causes a state transition are updated according to the length of path at that level, if the state transition is what was ordered at all lower levels. This restriction comes from the

constraint that super-managers should only learn from the fruits of the honest labour of sub-managers, ieonly if they obey their managers. A time-out is required if, for instance, a lower level manager is forced to try an impossible move, such as searching for the goal in the wrong region.
Feudal slower initially
Faster later

(d) Conclusions
Advantages

i. Learns more about environment than standard Q- Learning approach
ii. Structured exploration

Costs

i. Information hiding may introduce inefficiencies
ii. Submanagers learn solutions to subproblems, even if these are not relevant to goal
iii. Task may appear non- markovian at high level abstraction

5. Markov Options. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning Sutton et al (1999)

(a) Semi-Markov Decision Processes (SMDP)
MDP : Amount of time between decisions fixed
SMDP: Amount of time between decisions is random variable (T)
- Continuous
- Discrete
Treat system as waiting for T periods
Instantaneous state transition afterward

(b) Effectively need to marginalize over all T

(c) Bellman optimality equations
V-V

$$v_*(s) = \max_a [R_s^a + \sum_{s',t} \gamma^t P(s', t|a, s) v_*(s')]$$

Q-V

$$q_*(s, a) = R_s^a + \sum_{s',t} \gamma^t P(s', t|a, s) \max_{a'} q_*(s', a')$$

(d) Control via V, Q-learning again

(e) Markov Options Formalization. e.g., "Open-the-door"
Option (O)
Input set $I \subset S$. Set of states where option O is available
Policy $\pi(s, a)$. Distribution of actions taken by options
Termination Condition. $\beta(s)$. Probability option ends in a given state.

(f) Assumptions:
Actions of core MDP- Primitive actions or one-step options $\beta(s) = 1$ for all $s \in S$
At least one option available in all states (actions are special case of options)
Option available in all states where it may continue

(g) Semi-Markov Options

"Sometimes it is useful for options to timeout, to terminate after some period of time has elapsed even if they have failed to reach any particular state. This is not possible with Markov options because their termination decisions are made solely on the basis of the current state, not on how long the option has been executing. To handle this and other cases of interest we allow semi-Markov options, in which policies and termination conditions may make their choices dependent on all prior events since the option was initiated."
Options where actions may depend on entire history of observations, since beginning of option (but not events prior to s_t)
$\mu : S \times \cup_{s \in S} O_s \rightarrow [0, 1]$
Allow options that terminate after fixed number of steps
Allow policies over options
" Semi-Markov options also arise if options use a more detailed state representation than is available to the policy that selects the options, as in hierarchical abstract machines [52,53] and MAXQ [16]. Finally, note that hierarchical structures, such as options that select other options, can also give rise to higher-level options that are semi-Markov (even if all the lower-level options are Markov). Semi-Markov options include a very general range of possibilities."
"Our definition of options is crafted to make them as much like actions as possible while adding the possibility that they are temporally extended. Because options terminate in a well defined way,we can consider sequences of them in much the same wayas we consider sequences of actions. We can also consider policies that select options instead of actions, and we can model the consequences of selecting an option much as we model the results of an action. Let us consider each of these in turn."

(h) "More interesting for our purposes are policies over options. When initiated in a state s_t, the Markov policy over options $\mu : S \times O \rightarrow [0, 1]$ selects an option $o \in O_{s_t}$ according to probability distribution $\mu(s_t, \cdot)$. The option o is then taken in s_t, determining actions until it terminates in s_{t+k}, at which time a new option is selected, according to $\mu(s_{t+k}, \cdot)$, and so on. In this way a policy over options, μ, determines a conventional policy over actions, or flat policy, $\pi = flat(\mu)$."
Flat policies
- Policy over primitive actions of core MDP
- All μ correspond to some flat policy flat(μ)
- Typically Non-Markovian even when all policies are Markovian

(i) These ideas lead to natural generalizations of the conventional value functions for a given policy.

$$V^{\mu}(s) := \mathbb{E} r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | E(\pi, s, t)$$

where $E(\pi, s, t)$ denotes the event of π being initiated in s at time t.

(j) "Action-value functions generalize to option-value functions. We define $Q^{\mu}(s, o)$, the value of taking option o in state $s \in I$ under policy μ, as

$$Q^{\mu}(s, o) := \mathbb{E} r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | E(o\mu, s, t)$$

where $o\mu$, the composition of o and μ, denotes the semi-Markov policy that first follows o until it terminates and then starts choosing according to μ in the resultant state."

(k) SMDP Q-learning
State-option reward

$$R(s,o) := \mathbb{E}r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | E(o,s,t)$$

Transition function

$$p(s'|s,o) = \sum_t \gamma^t P(s',t)$$

Optimality equations: Similar to Bellman.

(l) Intra-option learning methods - Learn online while option executes

(m) Add temporally-extended activities without precluding fine-grained control
Exclude some primitives
- Restrict set of learnable policies
- Increase efficiency, prevent flailing
Utilize options to achieve subgoals
- Define subgoal-specific reward functions and use for option policy
- Set options to terminate upon subgoal completion

5.3.13 Task-agnostic RL

1. Take away reward function from RL. What happens?
   In many RL tasks, at the beginning you are just exploring. How can you make this phase more efficient? Or, suppose you want to learn many rewards, how?
2. Internal reward predicates
   Deepmind control. Want closed loop control. Collect and infer.
   Infer: off-policy algorithms.
   NFQ algorithm (Riedmiller, 2005)
   Collect: sample meaningful solutions.
   In the absence of external rewards, do something reasonable to explore
   Learning by playing: SAC-X (Riedmiller et al 2018)
   Given: external task rewards and internal reward predicates (IRP) Principles: - store transitions - off-line off-policy - distillation - meta- learn
   IRP: principle: change distributions of sensor values
   eg. activate touch sensor, relation between objects (robust against these, want them to be transferable)
   Example: cleanup tasks, ball-in-cup
   Alternatives: reward shaping, demonstrations (hard to do in practice)
   Grand challenge: for general AI, learn the sub-rewards on its own.
3. Goal-conditioned policies
4. State marginal matching
5. Chelsea Finn. How to learn from unlabeled interactions?
   We are good at single task. how about many tasks?
   1. Learn Task Agnostic Models
   - Collect unlabeled robot data
   - Learn visual predictive model
   - Plan predictively
   - Execute
   2. Learn Task Agnostic Repesentation
   3. Learn Task Agnostic Models for Complex tasks

6. Doina Precup: Generalized Value Functions
   Building knowledge:
   Procedural knowledge
   Predictive, empirical knowdedge
   Knowledge must be: expressible, learnable,
   in Procedural knowledge: options.
   In predictive knowledge: classically, value function.
   Like it: bootstrapping, sampling
   GVF

$$v_{\pi,c,\gamma} = \mathbb{E} \sum_{k \geq t} c(S_k, A_k, S_{k+1}) \prod_{i=t+1^k} \gamma(S_t) | S_t = s, A$$

   use cumulants, continuation functions
   How to evaluate?
7. Multi-task RL: assume latent variables shared across learning tasks
   CAVIA - Hoffman et al - reconstruct image from pixels
8. Dynamic model for RL: State prediction
   Priors: physics, language

5.4 Other topics

1. Instruction following
2. Reward function shaping
3. Robustness/safety
4. Inverse rl
5. Instruction conditional rl
6. Imitation learning
7. Curiosity and intrinsic motivation

6 Special topics

6.1 Adversarial examples

- Neural networks are vulnerable to adversarial examples. An adversarial example is one that is designed to "fool" the machine learning system, leading to mistakes. (Szegedy et al., 2013; Biggio et al., 2013),
  Tutorial: https://adversarial-ml-tutorial.org/. Notes, in Jupyter notebook format.
  Video: https://www.youtube.com/watch?v=TwP-gKBQyic.
  Course: http://www.crystal-boli.com/teaching.html
  Cleverhans. Library of attacks.
  Website: robust-ml.org/.
- There are several models for adversarial attacks: white-box attacks assume the attackers know the model, while black-box attackers do not.

- How can we learn models robust to adversarial inputs?
- Let x be the training example, $f(x) := f(x; W)$ be the model, and L be the loss. A simple way to create an adversarial example out of x is to maximize the loss subject to a small perturbation. We can solve the problem

$$\max_{\delta} L(f(x+\delta))$$
$$s.t. |\delta|_\infty \leq \varepsilon$$

  using projected gradient descent (PGD).
- Sometimes, even a much smaller class of perturbations is enough. E.g., rotating and translating an image can lead to adversarial examples. Data augmentations helps a bit, but does not fully solve it. (Fawzi Frossard 2015, Engstrom Tran Tsipras Schmidt Madry 2018)
- Many defenses have been devised to protect from this attack. Several of them rely on "perturbing" the training example that the adversary hands to them, so that the adversary cannot be "too specific" in devising its attack.

  For instance, randomized defenses apply a random transform $x \to t(x)$, so that $t \sim T$ before classifying.
- A unifying perspective using robust optimization has been proposed by Madry et al, 2017 "Towards Deep Learning Models Resistant to Adversarial Attacks"

  Robust optimization references: Stoyer 1973, Ben-Tal et al 2011, Xu and Manor, 2009.

  They define the adversarial risk minimization problem

$$\min_W \rho(W) := \mathbb{E}_{(x,y)\sim D} \max_{\delta \in S} L(W, x+\delta, y)$$

  Here D is the data distribution.

  Here now L has different parameters, namely the weights W, the feature vector x, and the label y. Also S is a set of allowed perturbations.

  "Formulations of this type (and their finite-sample counterparts) have a long history in robust optimization, going back to Wald"

  They view the saddle point problem as the composition of an inner maximization problem and an outer minimization problem.

  1. Attack: Simplest one-step. Fast Gradient Sign Method (FGSM) (Goodfellow et al, explaining and harnessing adversarial examples)

$$x + \varepsilon \operatorname{sign}(\nabla_x L(W, x, y)).$$

     One-step approx of maximizing inner term. Where does this come from? Take Δ to be the ℓ_∞ ball of radius ε, so the projection is $P_\Delta(\delta) = Clip(\delta, [-\varepsilon, \varepsilon])$. As the step size goes to infinity, we always reach the corner of the box, so that the second constraint is active, and thus we get FGSM.
     Iterative methods: FGSM^k, closely related to projected gradient descent (PGD) on the negative loss function (Kurakin et al, Adversarial machine learning at scale).

$$x^{t+1} = \prod_{x+S} [x^t + \varepsilon \operatorname{sign}(\nabla_x L(W, x^t, y))]$$

     Note that vanilla PGD for $\max_{\delta\in\Delta} L(W, x+\delta, y)$ is:

$$\delta^{t+1} = \prod_{\Delta}[\delta^t + \varepsilon \nabla_\delta L(W, x + \delta^t, y)]$$

Aside: Gradients often small exactly at data points, so don't use "standard" PDG. Instead

$$\delta^{t+1} = \prod_{\Delta}[\delta^t + \arg\max_{\|v\| \leq \varepsilon} v^\top \nabla_\delta L(W, x + \delta^t, y)]$$

E.g., for ℓ_∞ (typical to choose inner norm the same as the Δ constraint), we get $\arg\max_{\|v\| \leq \varepsilon} v^\top \nabla_\delta L(W, x + \delta, y) = \varepsilon \operatorname{sign}(\nabla_x L(W, x, y))$, reducing to FGSM.

2. Targeted attacks: Also possible to explicitly try to change label to particular class

$$\max_{\delta \in \Delta}[L(W, x + \delta, y) - L(W, x + \delta, y_{target})]$$

For multi-class cross entropy loss

$$L(W, x, y) = \log \sum_i \exp h_\theta(x)_i - h_\theta(x)_y$$

the normalization cancels and this reduces to

$$\max_{\delta \in \Delta}[h_\theta(x + \delta)_{target} - h_\theta(x + \delta)_y]$$

Most attacks are variants of PGD for different norm bounds. (1) Norm/perturbation (2) Optimization algorithm

3. These attack mechanisms are based on local search (lead to lower bound on objective). Alternatives:
Combinatorial optimization (exact)
Convex relaxation (upper bound). Will give us verification.
For linear models, you can solve these exactly, and all cases collapse to one.

$$\max_{\delta \in \Delta} L(\theta^\top (x + \delta) y) = L(\min_{\delta \in \Delta} \theta^\top (x + \delta) y) = L(\theta^\top x y - \varepsilon \|\theta\|_*)$$

4. Combinatorial optimization: exact. Consider a ReLU based feedforward net.

$$\begin{aligned} z_1 &= x \\ z_{i+1} &= ReLU(W_i z_i + b_i), \;\; i = 1, \ldots, d-1 \\ h_\theta(x) &= W_d z_d + b_d \end{aligned}$$

Targeted attach in ℓ_∞ norm can be written as the optimization problem

$$\begin{aligned} &\min_{z_{1:d}} (e_y - e_{y_{targ}})^\top (W_d z_d + b_d) \\ &s.t. z_{i+1} = ReLU(W_i z_i + b_i), \;\; i = 1, \ldots, d-1 \\ &\|z_1 - x\|_\infty \leq \varepsilon \end{aligned}$$

Problem: nonlinear equality constraint. Can write it equivalently as binary mixed integer program. Can use off-the-shelf solvers for small networks. One of the key

aspects of finding an efficient solution is to provide tight bounds on the pre-relu activations $l_i \leq W_i z_i + b_i \leq u_i$. How to do it? In general, if $l \leq z \leq u$, then

$$W_+ l - W_- u \leq Wz \leq W_+ u - W_- l$$

Loose in general, but useful for IP

5. Certifying robustness. If we solve our integer program for some target class, and the objective is positive, then this is a certificate that there exists no adversarial example for that target class. If the objective is positive for all target classes, this is a verified proof that there exists no adversarial example.
6. Convex relaxations. Replace the bounded ReLU constraints with their convex hull, let LP. Relax it even more (LP duality).
   Interval-based methods. Formulate optimization only considering bound constraints. Has analytical solution, looser but fast.
   These can be loose in general. However if we train specifically so that they can be used to certify, they can be better. However, slow.

- Adversarial training
  1. Defense: "the training dataset is often augmented with adversarial examples produced by FGSM. This approach also directly follows from their formulation when linearizing the inner maximization problem. To solve the simplified robust optimization problem, we replace every training example with its FGSM-perturbed counterpart. More sophisticated defense mechanisms such as training against multiple adversaries can be seen as better, more exhaustive approximations of the inner maximization problem."
  2. "While there are many local maxima spread widely apart within $x_i + S$, they tend to have very well-concentrated loss values. This echoes the folklore belief that training neural networks is possible because the loss (as a function of model parameters) typically has many local minima with very similar values."
  3. "robustness against the PGD adversary yields robustness against all first-order adversaries" ???
     Observed empirically, even if we use attacks that are different from the exact training mechanism. e.g., run PGD iteration longer
  4. Descent Directions for Adversarial Training
     "A natural way of computing the gradient of the outer problem, $\nabla_W \rho(W)$, is computing the gradient of the loss function at a maximizer of the inner problem. This corresponds to replacing the input points by their corresponding adversarial perturbations and normally training the network on the perturbed input. A priori, it is not clear that this is a valid descent direction for the saddle point problem. However, for the case of continuously differentiable functions, Danskin's theorem - a classic theorem in optimization - states this is indeed true and gradients at inner maximizers corresponds to descent directions for the saddle point problem."

     $$\nabla_\theta \max_{\delta \in \Delta} L(W, x + \delta, y) = \nabla_\theta \max_{\delta \in \Delta} L(W, x + \delta^*, y)$$

     where $\delta^* = \arg\max_{\delta \in \Delta} L(W, x + \delta, y)$.
  5. Network Capacity and Adversarial Robustness
     "For a fixed set S of possible perturbations, the value of the problem is entirely dependent on the architecture of the classifier we are learning. Consequently, the architectural capacity of the model becomes a major factor affecting its overall performance. At a high level, classifying examples in a robust way requires a stronger

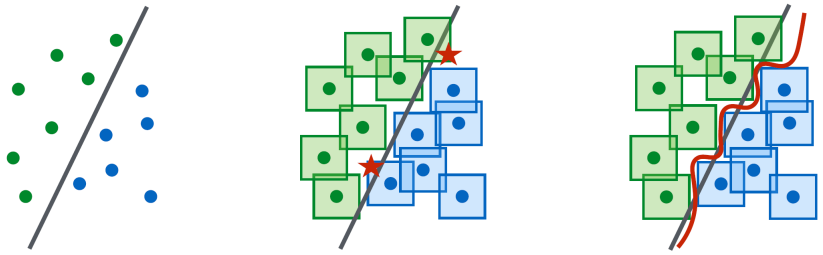


Figure 84: A conceptual illustration of natural vs. adversarial decision boundaries.

classifier, since the presence of adversarial examples changes the decision boundary of the problem to a more complicated one (see Figure 84 for an illustration)."

6. Adversarial training
   (a) For fixed W, perform PGD on randomly chosen datapoints x, starting from randomly chosen perturbation. Get adversarial perturbations $\delta^*(x)$
   (b) Perform one step of GD on W.

$$W \leftarrow W - \alpha/|B| \sum_{x \in B} \nabla_W L(W, x + \delta^*(x), y)$$

- However, in the high-profile paper "Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples", (Athalye et al, ICML 2018), it was shown that many defenses relying on "obfuscated" (transformed) gradients can be circumvented.

  For instance, suppose that the defense mechanism is to transform the input using a random transform $t \sim T$. They propose to attack this using

$$\max_{\delta} L(\mathbb{E}_{t \sim T} f(t(x + \delta)))$$
$$s.t. |\delta|_\infty \leq \varepsilon$$

  They show how to solve this using PGD. The key insight is that gradients can still be computed, because $\nabla \mathbb{E} f(t(x)) = \mathbb{E} \nabla f(t(x))$. Moreover, in practice they approximate the expectation by Monte Carlo sampling.

- In some cases, minimizing the adversarial loss is equivalent to (or can be upper bounded by) minimizing the usual loss on a different (regularized) class of functions. (Khim, Loh 2018)

- ML via Adv Rob lens.

  1. Do robust deep networks overfit? Empirically yes.
     Adv Rob Generalization needs more data
     Thm [Schmidt Santukar Tsipras Talwar Madry 2018]. Sample complexity of adv robust generalization can be significantly larger than that of standard generalization. Specifically, there is a d-dim distribution, st a single sample is enough to get an accurate classifier (with 0.99 accuracy), but need $\Omega(\sqrt{d})$ samples for better-than-chance robust classifier.

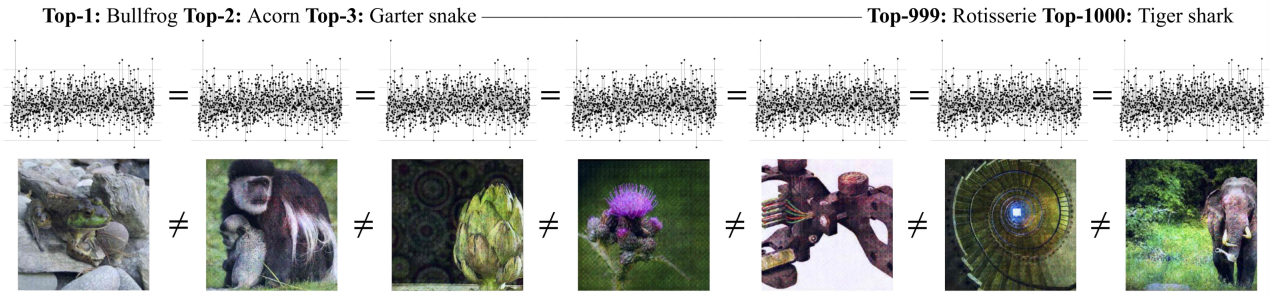


Figure 1: All images shown cause a competitive ImageNet-trained network to output the *exact same* probabilities over all 1000 classes (logits shown above each image). The leftmost image is from the ImageNet validation set; all other images are constructed such that they match the non-class related information of images taken from other classes (for details see section 2.1). The excessive invariance revealed by this set of adversarial examples demonstrates that the logits contain only a small fraction of the information perceptually relevant to humans for discrimination between the classes.

Figure 85: EXCESSIVE INVARIANCE CAUSES ADVERSARIAL VULNERABILITY.

2. Does being robust help standard generalization?
   Thm [Tsipras Santukar Engstrom Turner Madry 2018]. No free lunch: can exist a tradeoff between accuracy and robustness. Basic intuition: Must avoid weakly correlated features. In standard training, all estimable correlation is good. [So is lasso more robust?]
3. Unexpected benefits: Models become more semantically meaningful. Saliency map example. Also nearest "bird" to a "mammal" looks more like a bird.
   Robust models. Restricted GAN?

- Current challenges [2018 NeurIPS tutorial]. Embrace more of a worst-case mindset.
  1. Algorithms: Faster robust training + verification. Smaller models, new architectures?
  2. Theory: Adv robust generalization bounds, new regularization techniques
  3. Data: new datasets and more comprehensive set of perturbations
- EXCESSIVE INVARIANCE CAUSES ADVERSARIAL VULNERABILITY, ICLR 2019, (Jacobsen, Behrmann, Zemel, Bethge)

  "We show deep networks are not only too sensitive to task-irrelevant changes of their input, as is well-known from ε-adversarial examples, but are also too invariant to a wide range of task-relevant changes, thus making vast regions in input space vulnerable to adversarial attacks."

  Constuct invertible/bijective network

  $x \to z := F(x)$

  Such that $z = (z_s, z_n)$, and class labels are based on logits of z_s, while z_n is nuisance.

  Construct metameric examples $x_m = F^{-1}(z_s, \tilde{z}_n)$, by taking signal and nuisance from two different examples.

  Observe that the nuisance dominates. Visually the examples look like the nuisance class.

"We show such excessive invariance occurs across various tasks and architecture types. On MNIST and ImageNet one can manipulate the class-specific content of almost any image without changing the hidden activations."

Perturbation robust models are more vulnerable to invariance based adversarial examples.

Fix: Independence Cross-Entropy. Instead of just $\max I(y; z_s)$, maximize $I(y; z_s|z_n)$.

Note: these examples seem much harder to come up with than other adversarial examples, because they require a very special architecture (invertible, disentangled).

6.2 Neural ODEs

- There is a natural limit of residual networks of the form

$$z(t+1) = z(t) + f(z(t), t, W)$$

namely an ODE

$$\frac{dz(t)}{dt} = f(z(t), t, W)$$

- In the "Neural ODEs" paper (NeurIPS 2018), it is shown how to train a model of this form, to minimize a loss function $L(z(t_1))$ at some time t_1.

$$L(z(t_1)) = L(\int_{t_0}^{t_1} f(z(t), t, W)dt)$$

- The forward pass of the algorithm starts with some fixed $t_0, W, z(t_0)$, and solves the ODE up until t_1, to compute $z(t), f(z(t))$ for all $t \leq t_1$
- Next, we need to train the model. The key is how to backprop the gradients in continous time. They introduce the *adjoint* (which is well known in optimal control)

$$a(t) = \frac{\partial L}{\partial z(t)}$$

which is the derivative of the loss with respect to the hidden state $z(t)$ at time t.

- It turns out that there is another ODE for the adjoint:

$$\frac{da(t)}{dt} = -a(t)^\top \frac{f(z(t), t, W)}{\partial z}$$

Here the gradient is with respect to the first variable of f. Using this, we can start with $a(t_1) = L'(z(t_1))$, and solve backward in time for $a(t)$. Notice that this requires knowing $z(t)$ for all t, which we have from the forward pass (or we can recompute backwards in time).

- It turns out that computing the gradients wrt the parameters W reduces to the formula

$$\frac{dL}{dW} = \int_{t_0}^{t_1} a(t)^\top \frac{f(z(t), t, W)}{\partial W} dt$$

This can also be computed in the same way as above. Moreover, all computations can be put together into a single call to an ODE solver.

6.3 Physics informed NNs (PINNs)

- Standard NNs do not use well-established physical principles and laws (conservation of momentum etc)
- PINNs (Raissi, Perdikaris, Karniadakis, 2017) incorporate these. Let $u := u(t, x)$ be a physical system we want to model, e.g., heat or fluid flow. It is defined for $t \in [0, T]$ and for a region $x \in \Omega \subset \mathbb{R}^D$. The physical law governing it is a PDE

$$u_t + \mathcal{N}[u] = 0$$

  where $\mathcal{N}[\cdot]$ is a nonlinear differential operator.
- Consider the continuous time case. Let us assume that we have initial value data u^i, measured at location x_u^i and time t_u^i.

  Let us also denote $f := u_t + \mathcal{N}[u]$. Let t_f^i, x_f^i denote some collocation points for f, where the errors will be evaluated.

  The goal of the training will be to construct a solution u that minimizes both the initial value MSE

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2$$

  as well as the overall MSE at the collocation points

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2$$

- Now, u is parametrized by a suitable NN model $u(x, t; W)$ and we fit this to the data by minimizing a weighted combination $MSE_u + c \cdot MSE_f$.

  "In all benchmarks considered in this work, the total number of training data Nu is relatively small (a few hundred up to a few thousand points), and we chose to optimize all loss functions using L-BFGS; a quasi-Newton, fullbatch gradient-based optimization algorithm [17]. For larger data-sets a more computationally efficient mini-batch setting can be readily employed using stochastic gradient descent and its modern variants [18, 19]. Despite the fact that there is no theoretical guarantee that this procedure converges to a global minimum, our empirical evidence indicates that, if the given partial differential equation is well-posed and its solution is unique, our method is capable of achieving good prediction accuracy given a sufficiently expressive neural network architecture and a sufficient number of collocation points Nf ."

6.4 Information bottleneck and invariance

- "The Information Bottleneck (IB) was introduced by Tishby, Pereira, and Bialek (The information bottleneck method 1999) as a generalization of minimal sufficient statistics that allows trading off fidelity (sufficiency) and complexity of a representation. In particular, the IB Lagrangian reduces finding a minimal sufficient representation to a variational optimization problem."

  "Later, Tishby and Zaslavsky (2015) and Shwartz-Ziv and Tishby (2017) advocated using the IB between the test data and the activations of a deep neural network, to study the sufficiency and minimality of the resulting representation. In parallel developments, the IB Lagrangian was used as a regularized loss function for learning representation, leading

to new information theoretic regularizers (Achille and Soatto, 2018; Alemi et al., 2017a; Alemi et al., 2017b)."

- Achille & Soatto, "Emergence of Invariance and Disentanglement in Deep Representations", 2018, apply this theory. They show that invariance to nuisance factors in a deep neural network is equivalent to information minimality of the learned representation, and that stacking layers and injecting noise during training naturally bias the network towards learning invariant representations
- Setup: Training set $D = (x, y)$, where $x = \{x^{(i)}\}_{i=1}^n$, and $y = \{y^{(i)}\}_{i=1}^n$ is a collection of randomly sampled data points and their labels. They are sampled iid from a distribution $p_\theta(x, y)$. This is a Bayesian setup, and we assume a prior $\theta \sim p(\theta)$.
  Test datum x. Task: predict y
- Information theory:
  1. Shannon entropy: $H(x) = -\mathbb{E}_p \log p(X)$
  2. Conditional entropy: $H(x|y) = \mathbb{E}_{\bar{y}} H(X|y = \bar{y}) = H(x, y) - H(y)$
  3. Mutual information: $I(x; y) = H(x) - H(x|y) = H(x) + H(y) - H(x, y)$
  4. Conditional mutual information: $I(x; y|z) = H(x|z) - H(x|y, z)$
     Can see that $I(x; y|z) = H(x, z) + H(y, z) - H(x, y, z) - H(z)$
  5. Kullback-Leibler (KL) divergence: $KL(p||q) = \mathbb{E}_p \log p(X)/q(X)$
  6. Cross-entropy: $H_{p,q} = -\mathbb{E}_p \log q(X) = KL(p||q) + H(p)$
  7. Total correlation (TC), aka multivariate mutual information:
     $$TC(z) = -KL(p||\prod_i p_i)$$
     where p_i are the marginal distributions of the components of p. Recall that the KL divergence between two distributions is always non-negative and zero if and only if they are equal. In particular $TC(p(z))$ is zero if and only if the components of z are independent, in which case we say that z is disentangled.
  8. Identity:
     $$I(z; x) = \mathbb{E}_{x\sim p} KL(p(z|x)||p(z))$$
  9. Markov Chain: We say that x, z, y form Markov chain, indicated with $x \to z \to y$, if $p(y|x, z) = p(y|z)$
  10. Data Processing Inequality (DPI) for a Markov chain $x \to z \to y$:
      $$I(x; z) \geq I(x; y)$$
- Representations: z is a representation of x if z is a stochastic function of x, or equivalently if the distribution of z is fully described by the conditional $p(z|x)$ (so we have a Markov chain $y \to x \to z$)
  1. z is sufficient for y if $y \perp\!\!\!\perp x|z$, or $I(z; y) = I(x; y)$. "z contains as much information about y as does x"
     [Note: the classical definition of sufficiency is different. It states that a statistic $t(x)$ is sufficient for a parameter θ (not an observed quantity), if the distribution $x|t(x)$ does not depend on θ]
     Note that $I(z; y) = I(x; y)$ is equivalent to $H(y) - H(y|z) = H(y) - H(y|x)$, or $H(y|z) = H(y|x)$
     Explicitly, this can also be written as $H(y, z) - H(z) = H(y, x) - H(x)$
     Note that we always have $I(z; y) \leq I(x; y)$, or equivalently
     $H(y|z) \geq H(y|x)$
     with equality when z is sufficient for y.

2. z is minimal for y if $I(z;x)$ is smallest among sufficient representations

- Information Bottleneck (IB) Lagrangian

$$L(p(z|x)) = H(y|z) + \beta \cdot I(z;x)$$

Tradeoff between

1. $H(y|z)$: small when z contains a lot of information about y. Minimized when z is sufficient for y
2. $I(z;x)$: small when z contains only little information related to x. Minimized when z is independent of x (if deterministic, then constant)

Define a representation z by minimizing IB, as

$$\min_z L(p(z|x))$$

- Nuisance n: $y \perp\!\!\!\perp n$, or $I(y;n) = 0$.
  Representation z is invariant to nuisance if $z \perp\!\!\!\perp n$
  Representation z is maximally insensitive to nuisance if it is sufficient and minimizes $I(z;n)$. In the future this will be called invariant.
  This is more general than invariance of deterministic representations $z = f(z)$ to the action of a group g, in the form $f(gx) = f(x)$ [allows non-determinism, non-group]
  Proposition: For discrete y there is a nuisance such that $x = f(y,n)$
- Optimal representation z of x with respect to y. [Goals of representation learning can be expressed in terms of information theory!]
  1. z is sufficient for y
  2. minimal: minimize $I(z;x)$
  3. invariant $I(z;n) = 0$.
  4. maximally disentangled: minimize $TC(z)$
- Result: (Invariants from the Information Bottleneck) Minimizing the IB La- grangian in the limit $\beta \to 0$, yields a sufficient invariant representation z of the test datum x for the task y.
  i.e., Minimizes $H(y|z)$, and minimizes $I(z;n)$.

6.5 Gradient-based optimization

1. A big part of this presentation is borrowed from Sebastian Ruder. Notes prepared originally by Matteo Sordello.
2. Expected vs. Empirical Loss
   - $(x_i, y_i) \in \mathbb{R}^{d_x} \times \mathbb{R}$ for $i = 1, ..., n$, we call x the input and y the output.
   - $f(\theta, x, y)$ is the loss (risk) function. Two popular examples
     - Linear regression: $f(\theta, x, y) = (y - x \cdot \theta)^2$
     - Logistic regression: $f(\theta, x, y) = \log(1 + e^{-yx \cdot \theta})$
   - The **expected** loss is
   $$L(\theta) = \mathbb{E}_{(x,y)}[f(\theta, x, y)]$$
   - The **empirical** loss is
   $$L_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} f(\theta, x_i, y_i) := \frac{1}{n} \sum_{i=1}^{n} f_i(\theta)$$

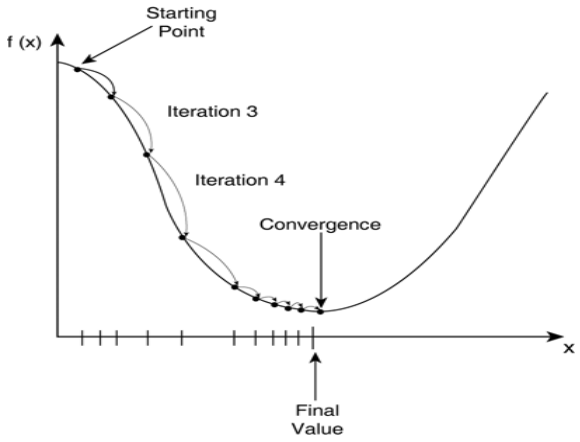
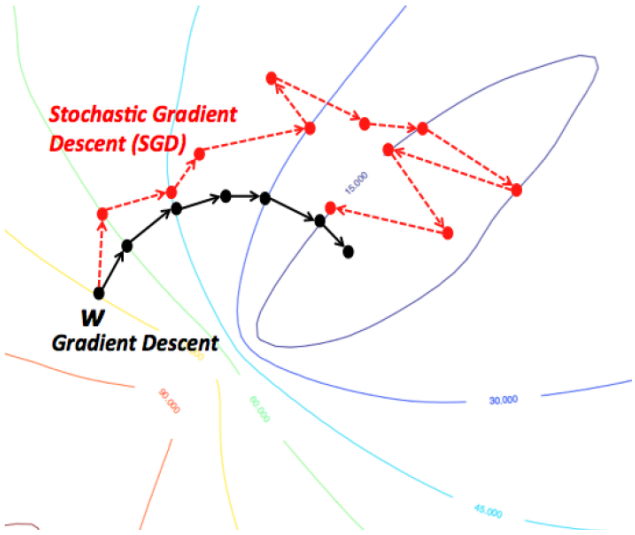


Figure 86: Gradient descent

3. Gradient descent. Figure 86
   - Gradient descent is a way to minimize an objective function $F(\theta)$.
     - $\theta \in \mathbb{R}^d$ are the model parameters
     - η_t is the learning rate at iteration t.
     - $\nabla_\theta F(\theta)$ is the gradient of the objective function with respect to the parameters
   - The parameters get updated in the **opposite** direction of the gradient, following the equation
   $$\theta_{t+1} = \theta_t - \eta_t \cdot \nabla_\theta F(\theta_t)$$
4. Variants
   - Batch gradient descent
   - Stochastic gradient descent
   - Mini-batch gradient descent

   They only differ in the amount of data used for each update. Figure 87, 88
5. Batch gradient descent. At every iteration we compute the gradient using the whole dataset. The update equation is
   $$\theta_{t+1} = \theta_t - \eta_t \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta f_i(\theta_t)$$
   - **Pros**: Guaranteed to converge to global minimum for convex error surfaces and to a local minimum for non-convex surfaces.
   - **Cons**:
     - Very slow
     - Intractable for datasets that do not fit in memory
     - No online learning
6. Stochastic gradient descent. We compute an update for each data point. The update equation is
   $$\theta_{t+1} = \theta_t - \eta_t \cdot \nabla_\theta f_{i_t}(\theta_t)$$
   where $i_t \in \{1, .., n\}$.

Figure 87: Batch gradient descent vs. SGD fluctuation (Source: Wikipedia)



- **Pros**:
  - Much faster than batch gradient descent
  - Allows online learning
- **Cons**: High variance updates.
- SGD shows same convergence behaviour as batch gradient descent if learning rate is slowly **decreased** over time.

7. Mini-batch gradient descent. At every gradient update a mini-batch of b data points is used. Let B be the set indices, the update equation is

$$\theta_{t+1} = \theta_t - \eta_t \cdot \frac{1}{b} \sum_{i \in B} \nabla_\theta f_i(\theta_t)$$

   - **Pros**: Reduces variance of updates
   - **Cons**: Mini-batch size is a hyperparameter. Common sizes are 50-256
   - This is the algorithm used in practice most of the times in real applications.

8. Challenges
   - Choosing a learning rate η_t (constant or decreasing? How fast?)
   - Updating features to different extent
   - Avoiding suboptimal minima
9. Gradient descent optimization algorithms
   - Momentum
   - Nesterov accelerated gradient
   - Adagrad
   - Adadelta

Figure 88: Tradeoffs

Method	Accuracy	Update Speed	Memory Usage	Online Learning
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Table: Comparison of trade-offs of gradient descent variants

Figure 89: Ravine

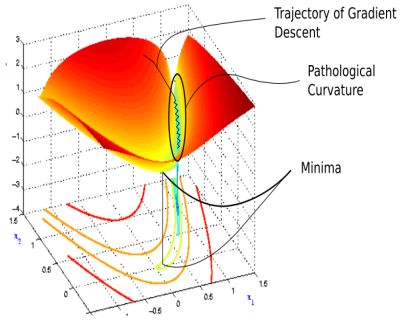


Figure 90: Ravine in optimization

- RMSprop
- Adam

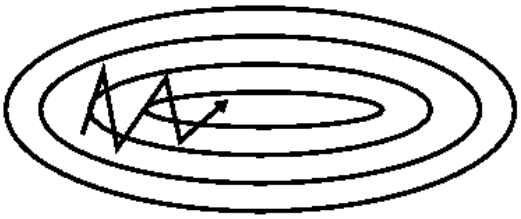
10. Ravines. A ravine is a slope landform of relatively steep (cross-sectional) sides. Figure 89
11. Momentum. Figure 91.
    - SGD has trouble navigating ravines
    - Momentum [Qian, 1999] helps SGD accelerate
    - A fraction γ of the update vector at step $t-1$ is added to the gradient

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta f(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

or equivalently $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta f(\theta_t) - \gamma(\theta_{t-1} - \theta_t)$

Figure 91: SGD without momentum

Figure 92: SGD with momentum



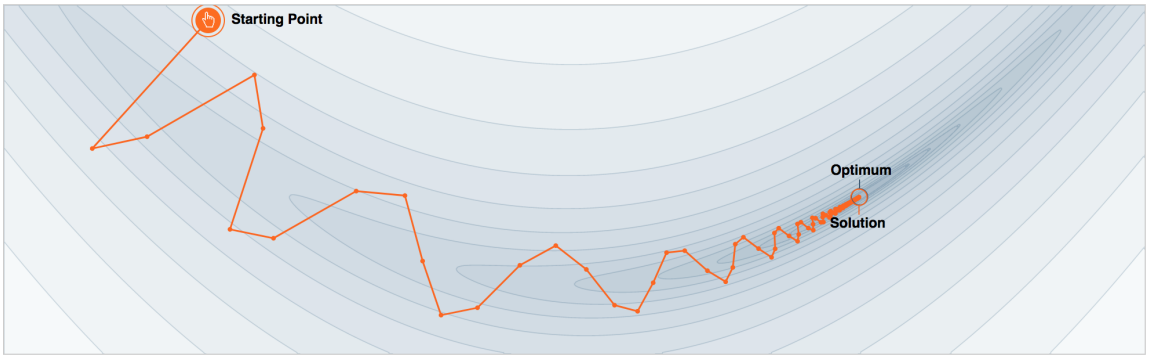


Figure 93: Optimization with momentum (Source: distill.pub)

12. Advantages of momentum. Figure 93. IDEA: a ball rolling down a hill accumulating momentum
    - Reduces updates for dimensions whose gradients change directions
    - Increases updates for dimensions whose gradients point in the same directions
13. Nesterov accelerated gradient **Problem with momentum**: the ball has no notion of where it's going
    - Momentum blindly accelerates down slopes: First computes gradient, then makes a big jump
    - Nesterov accelerated gradient (NAG) [Nesterov, 1983] first makes a big jump in the direction of the previous accumulated gradient $\theta_t - \gamma v_{t-1}$. Then measures where it ends up and makes a correction, resulting in the complete update vector.

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta f(\theta_t - \gamma v_{t-1})$$
$$\theta_{t+1} = \theta_t - v_t$$

14. Nesterov update. Figure 94
    - small blue vector: current gradient
    - brown vector: previous accumulated gradient
    - big blue vector: current accumulated gradient
    - red vector: gradient after the big jump

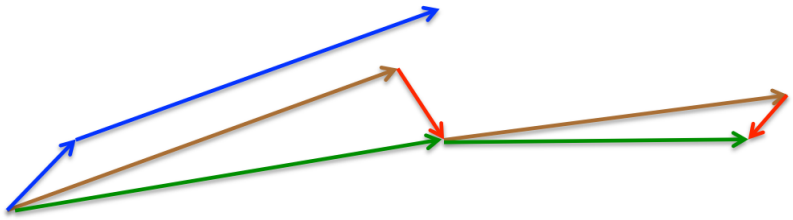
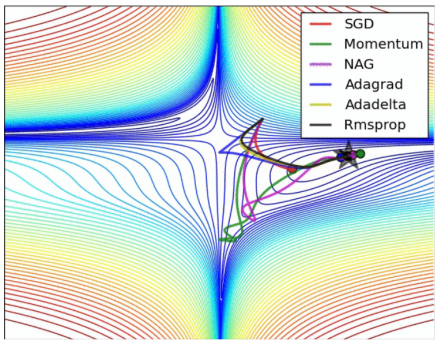


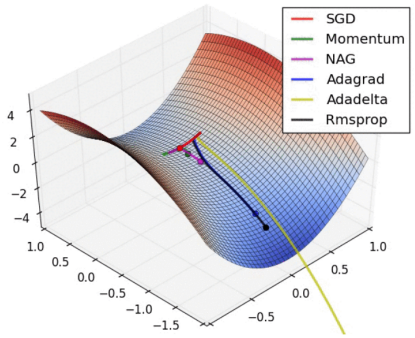
Figure 94: Nesterov

- green vector: NAG update

15. Adagrad **Notation**: $g_t = \nabla_\theta f(\theta_t)$.
    Problem: up to now the learning rate η_t was the same for each parameter.
    - Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (large updates for infrequent parameters, small updates for frequent parameters)
    - It divides the learning rate by the square root of the sum of squares of historic gradients. Its update is
    $$\theta_{t+1} = \theta_t - \frac{\eta_t}{\sqrt{G_t + \epsilon}} \cdot g_t$$
    where G_t is the diagonal matrix where each entry is the sum of the squares of the gradients up to time t.
    - ϵ is a smoothing term to avoid division by 0.
    - **Pros**:
        - Well-suited for dealing with sparse data
        - Significantly improves robustness of SGD
        - Lesser need to manually tune learning rate
    - **Cons**: Accumulates squared gradients in denominator. Causes the learning rate to shrink and become infinitesimally small
16. Adadelta **Idea**: only keep a window of accumulated past squared gradients (**inefficient**)
    - Defines running average of squared gradients $avg(g^2)_t$ at time t with
    $$avg(g^2)_t = \gamma \cdot avg(g^2)_{t-1} + (1-\gamma) g_t^2$$
    The parameter γ is similar to the momentum term, usually $\gamma = 0.9$.
    - The preliminary Adadelta upgrade is then
    $$\theta_{t+1} = \theta_t - \frac{\eta_t}{\sqrt{avg(g^2)_t + \epsilon}} \cdot g_t$$
17. Units matching with Adadelta. Note As well as in SGD, Momentum, or Adagrad, hypothetical units do not match. If $f(\theta)$ is unitless, then
    $$g = \frac{\partial f(\theta)}{\partial \theta} \propto \frac{1}{\text{units of } \theta} \quad \Rightarrow \quad \frac{\eta_t}{\sqrt{avg(g^2)_t + \epsilon}} \cdot g_t \quad \text{is unitless}$$
    - Define $\Delta\theta_t = \theta_{t+1} - \theta_t$ and a running average of squared parameter updates
    $$avg(\Delta\theta^2)_t = \gamma \cdot avg(\Delta\theta^2)_{t-1} + (1-\gamma)\Delta\theta_t^2$$



(a) SGD optimization on loss surface contours



(b) SGD optimization on saddle point

Figure 95: Animations

- The final Adadelta update is

$$\theta_{t+1} = \theta_t - \frac{\sqrt{avg(\Delta\theta^2)_{t-1} + \epsilon}}{\sqrt{avg(g^2)_t + \epsilon}} \cdot g_t$$

18. Comments on Adadelta
    - In the final formula we use $avg(\Delta\theta^2)_{t-1}$ instead of $avg(\Delta\theta^2)_t$ because it is not known
    - Adadelta uses the hyperparameters γ and ϵ, but it's **robust** so they don't need to be tuned
    - The learning rate η_t does not have to be specified!
19. RMSprop

    Developed independently from Adadelta around the same time by Geoff Hinton. Its updates are the same as in the preliminary Adadelta

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{\sqrt{avg(g^2)_t + \epsilon}} \cdot g_t$$

20. Some cool animation. Figure 95

    Link to animations
21. Adam (Adaptive Moment Estimation)
    - Like Adadelta and RMSprop, also Adam [Kingma and Ba, 2015] stores a running average of **past squared gradients**

$$avg(g^2)_t = \beta_2 \cdot avg(g^2)_{t-1} + (1 - \beta_2) g_t^2$$

    - Moreover, similar to Momentum, it also stores an exponentially decaying average of **past gradients**

$$avg(g)_t = \beta_1 \cdot avg(g)_{t-1} + (1 - \beta_1) g_t$$

They are estimates of the first and second moments of the gradient (hence the name). The parameters β_1 and β_2 are decay rates.

If initialized as vectors of zeros, $avg(g^2)_t$ and $avg(g)_t$ are biased towards zero. Compute bias-corrected first and second moment estimates

$$m_t = \frac{avg(g)_t}{1-\beta_1^t}$$
$$v_t = \frac{avg(g^2)_t}{1-\beta_2^t}$$

The Adam update is

$$\theta_{t+1} = \theta_t - \frac{\alpha_t}{\sqrt{v_t}+\epsilon} \cdot m_t$$

22. Why the bias correction

$$\begin{aligned}\mathbb{E}[avg(g^2)_t] &= \mathbb{E}[(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\cdot g_i^2] \\ &= \mathbb{E}[g_t^2]\cdot(1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i} + \zeta \\ &= \mathbb{E}[g_t^2]\cdot(1-\beta_2^t) + \zeta\end{aligned}$$

- $\zeta = 0$ if $\mathbb{E}[g_i^2]$ is stationary, otherwise it can be kept small assigning small weights to gradients too far in the past.

23. Convergence analysis
    - $f_t(\theta)$ is a sequence of convex cost functions of unknown nature
    - $\theta^* = \text{argmin}_\theta \sum_{t=1}^{T} f_t(\theta)$
    - The goal is to show that the regret

$$R(T) = \sum_{t=1}^{T}[f_t(\theta_t) - f_t(\theta^*)]$$

      is sublinear in T.

24. Convergence analysis

    Theorem 6.1. *Assume that the function f_t has bounded gradients $||\nabla f_t(\theta)||_2 \leq G$ and $||\nabla f_t(\theta)||_\infty \leq G_\infty$ for all $\theta \in \mathbb{R}^d$ and that the distance between any θ_t generated by Adam is bounded, $||\theta_n - \theta_m||_2 \leq D$ and $||\theta_n - \theta_m||_\infty \leq D_\infty$ for any $m, n \in \{1, ..., T\}$. Then Adam achieves the following guarantee, for all $T \geq 1$*

$$\frac{R(T)}{T} = O(\frac{1}{\sqrt{T}})$$

25. Experiment: Logistic Regression. Figure 96
    - stepsize is $\alpha_t = \alpha/\sqrt{t}$.
26. Experiment: Convolutional Neural Networks. Figure 97
27. Extensions: Adamax
    - Instead of L^2 norm we can consider the L^p norm

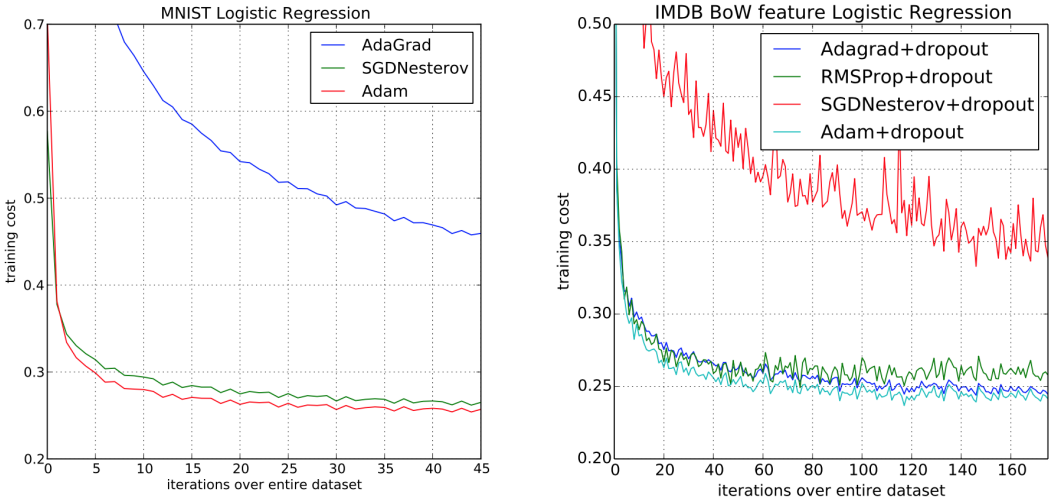


Figure 96: Logistic regression experiment

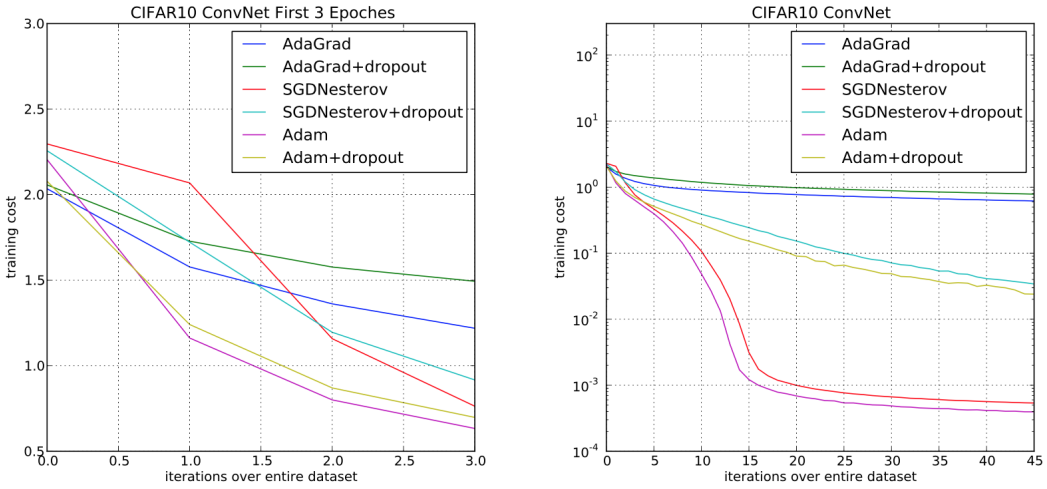


Figure 97: Experiment: Convolutional Neural Networks

- It is usually unstable, but for $p \to \infty$ the algorithm is surprisingly simple and stable. Define $v_t = \beta_2^p \cdot v_{t-1} + (1 - \beta_2^p) \cdot |g_t|^p$ and

$$u_t = \lim_{p \to \infty} (v_t)^{1/p} \quad \Rightarrow \quad u_t = \max(\beta_2 \cdot u_{t-1}, |g_t|_1)$$

- Note that u_t now does not need bias correction

The update rule for Adamax is

$$\theta_{t+1} = \theta_t - \frac{\eta_t}{u_t} \cdot m_t$$

28. Extensions: Temporal Averaging

The last iterate is noisy, instead one can use:

- Polyak-Ruppert averaging

$$\bar{\theta}_t = \frac{1}{t} \sum_{k=1}^{t} \theta_k$$

- running average

$$\bar{\theta}_t = \beta \cdot \bar{\theta}_{t-1} + (1 - \beta)\theta_t$$

starting from $\bar{\theta}_0 = 0$. Initalization bias can again be corrected by the estimator

$$\hat{\theta}_t = \frac{\bar{\theta}_t}{1 - \beta^t}$$

29. Additional Strategies

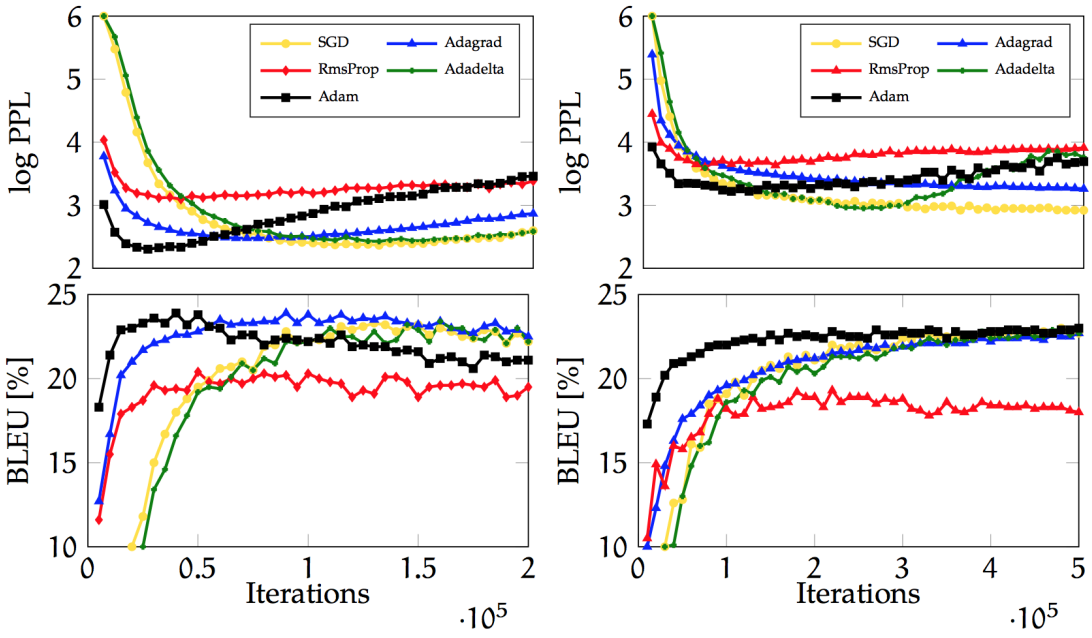
- **Nadam**: combine Adam and NAG modifying the momentum term m_t.
- **Hogwild!**: a way to perform SGD updates in parallel. It only works if the input data is sparse, as in this case it is unlikely that processors will overwrite useful information.
- **Early stopping**: "beautiful free lunch". Always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.
- **Gradient noise**: adding noise to each gradient update

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2) \qquad \text{where} \quad \sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

makes networks more robust to poor initialization and helps training particularly deep and complex networks. They use $\gamma = 0.55$.

30. Comparison on Neural Machine Translation

- The goal is to minimize the cross-entropy over the training samples
- Two tasks:
  - English → Romanian, 604K pairs of bilingual sentences with 16.8M English and 17.7M Romanian words
  - German → English, 4.2M sentence pairs with 133M German and 125M English words
- The hyperparameters are always the ones proposed in the original publications
- First single algorithm, then a combination of them
- How to evaluate performance
  - PPL: perplexity, strictly related to the entropy, we want to minimize it



(a) En→Ro *(b) De→En*

Figure 1: log PPL and BLEU score of all optimizers on validation sets.

Figure 98: NMT1

  - BLEU: bilingual evaluation understudy, measure of similarity between texts, we want to maximize it

31. Experiments. Figure 98, 99
32. Bibliography
    (a) Ruder S (2016). An overview of gradient descent optimization algorithms. *arXiv preprint* arXiv:1609.04747
    (b) Kingma DP, Ba J (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint*: arXiv:1412.6980
    (c) Bahar P, Alkhouli T, Peter JT, Brix CJS, Ney H (2017) Empirical Investigation of Optimization Algorithms in Neural Machine Translation. *The Prague Bulletin of Mathematical Linguistics*, 108, p. 13-25.

6.6 Design of new architectures - gradient computations

- One of the key aspects for the design of new architectures/algorithms is: how to computate gradients efficiently?
- Examples:
- Multilayer networks - backpropagation (chain rule)
- Neural ODEs - adjoint state method

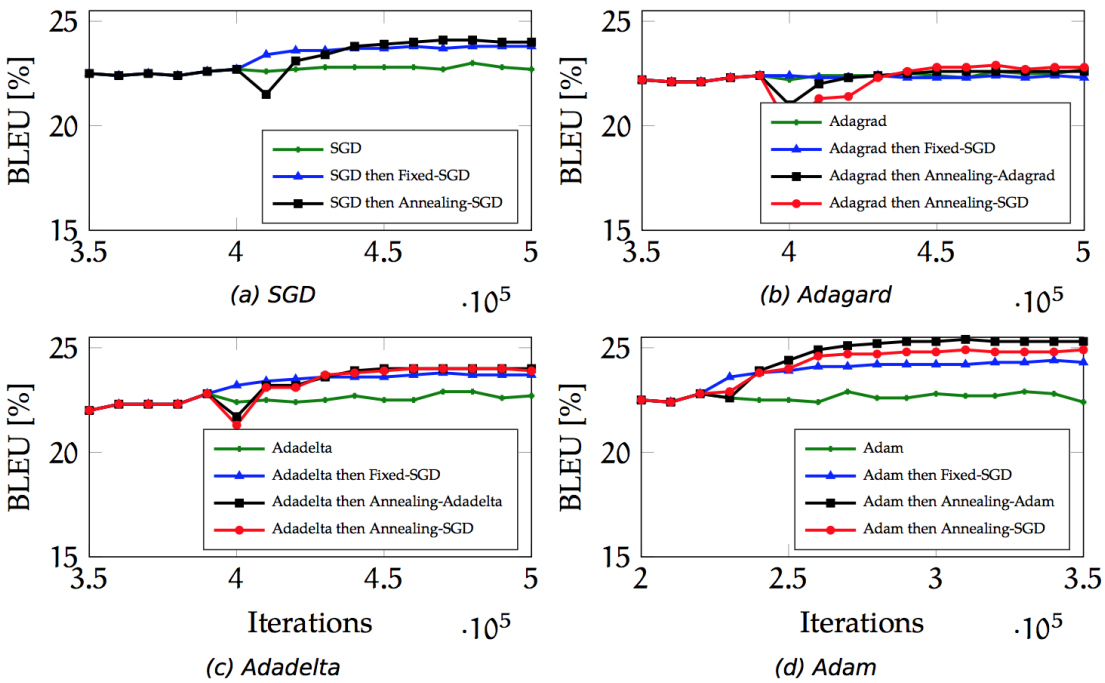


Figure 3: BLEU of optimizers followed by the combinations on the val. set for De→En. The representation of x-axis of Adam is different as it is faster.

Figure 99: NMT2

- GAN - alternating gradient updates (with expectation estimated by MC)
- Reinforcement learning:
  - Value function approximation - Approximate GD -¿ SGD -¿ replace value function by MC/TD estimate
  - Policy gradient - score function + MC
- Adversarial examples - obfuscated gradients
- Structured attention and predicion - regularization

  Differentiable Dynamic Programming for Structured Prediction and Attention Arthur Mensch, Mathieu Blondel

  Empirical successes in deep learning and modern computational infrastructure have inspired a push for end-to-end learning, but these efforts have largely been limited to learning models with stacked layers of elementary linear algebraic operations (due to the ease of automatic differentiation). The authors of this paper observe a growing desire for modeling pipelines that can process the optimal value and/or argument of a dynamic program (in domains like sequence prediction, time-series alignment, and machine translation).

  Unfortunately, the optimal value and argument of a dynamic program are typically non-differentiable, hence cannot be directly inserted as a layer of a neural network. There is a wealth of prior literature on probabilistic graphical models that instead approaches

dynamic programs by moving from the so-called (max, +) semiring to the (+, x) semiring; here, the sum-product algorithm (also known as belief propagation, message-passing, etc.) is the typical computational workhorse.

In this paper, the authors observe that existing probabilistic methods can be reframed as a particular instance of a general class of max operators with strongly convex regularizers (examples include entropy, l2 norm). Elementary results from convex duality show that this regularized max is smooth and differentiable (with the gradient as the maximizing argument). The authors go further first by showing that the optimal value and argument of the regularized max are differentiable. Next, they demonstrate that their gradients are explicitly computable. Finally, they prove bounds on the distance between the solutions to the regularized max and the original dynamic program.

6.7 Distributed training

1. Massive data creates new problems: how to process, analyze, learn from...
2. Example:
   - In 2014, Facebook reported storing 300 Petabytes (PB) of data. i.e., 300,000 Terabytes
   - Typical hard drive can store 1Tb...
   - So the data must be distributed over many computers
   - Compute locally, and communicate to get final answer
3. Big data considerations
   - Components
     - Storage: Hard disk
     - Memory: RAM, various levels of caching
     - Computation: CPU, GPU, TPU
     - Communication (within and between machines)
   - There is always a *bottleneck*—the slowest component
4. Grand challenges
   - Design efficient methods/algorithms for all major computational tasks
   - Scalable, resource-efficient (adaptive), easy to use (tuning-free), reliable and resilient, verifiable guarantees...
   - More specific: How can we do statistics/machine learning in this setting?
5. State of the field
   - Active area of research at large tech companies (Google, Facebook,...) and in the AI/ML community
   - Standard frameworks: MPI, MapReduce, Spark, GraphLab
   - Expected that this area will grow in the future.
6. MapReduce (Dean, Ghemawat, 2004). Figure 100
7. Current approach to stats/ML
   - Data parallelism: Distribute data over machines. The loss is a sum over training examples. Do iterative calculation (e.g., gradient descent), where compute gradient by summing over machines. Figure 101
   - Made efficient and reliable by e.g. Spark (Zaharia et al 2010)

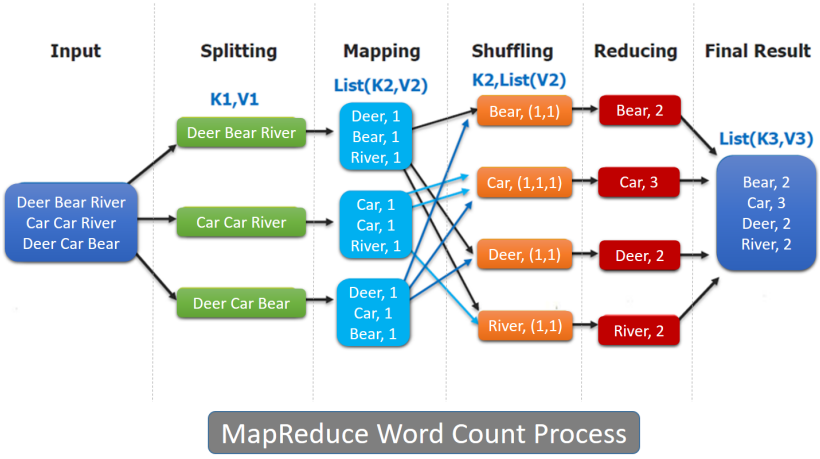


Figure 100: MapReduce

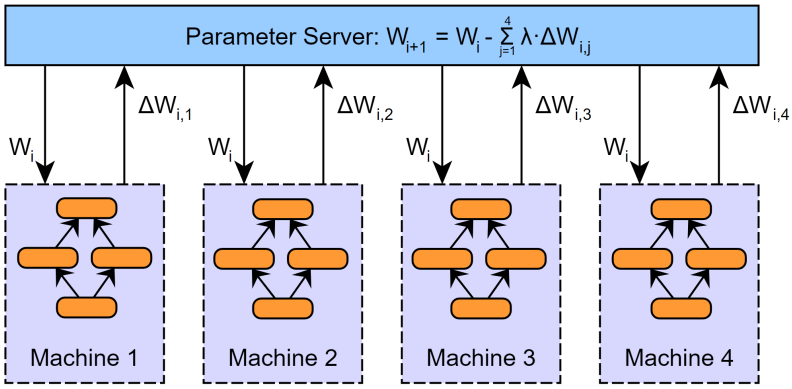
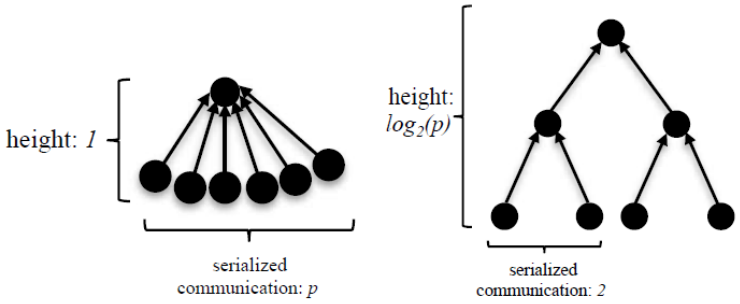


Figure 101: Distributed GD



(a) parameter server (b) reduction tree

Figure 3. Illustrating how parameter servers and reduction trees communicate weight gradients. In this figure, we only show the summing-up of weight gradients. We distribute the weight gradient sums by going back down the tree.

Figure 102: FireCaffe: near-linear acceleration of deep neural network training on compute clusters, Iandola et al 2016

8. Still not solved...
   - Communication often bottleneck
   - Various approaches: async, reduction tree (Figure 102)

6.7.1 Notes from Smola's class

http://alex.smola.org/teaching/berkeley2012/index.html
List of basic algorithms: all you need for startup

1. 1. Systems
   (a) Hardware
   cluster: lots of hardware failure
   can compute costs of various tasks: crawl web for $100,000k/month, using Amazon EC2
   (b) Data
   A lot of tracking going on
   (c) Parallelization
   Hashing: Argmin hash, Random caching tree - akamai
   More: gossip protocols, time sync/Byzantine fault tolerance,
   (d) Storage: raid error correcting code, parity
   (e) Processing: MapReduce
   (f) Databases: memcached. application, distributed GD - need to push/pull to memcached 2x??
2. 2. Basic statistics

(a) Naive Bayes: spam classifications + For exponential families, only need the sums of suffi stat - can compute

(b) Why need concentration inequalities - very convincing explanations + 100 users, sell ads for $1, failure 5% - how accurately do I estimate the revenue?
+ AB test for Chebyshev

3. 3. Sketches - Data streams
   - Time series (x_t, t)
   - Cash register x_t weighted, nonneg
   - Turnstile increase or decrease
   - Moments
4. 4. Optimization
   (a) Batch
   Very large dataset available
   Require parameter only at the end
   optical character recognition
   speech recognition
   image annotation / categorization
   machine translation

   (b) Online
   Spam filtering
   Computational advertising
   Content recommendation / collaborative filtering

   (c) Many parameters
   100 million to 1 Billion users Personalized content provision - impossible to adjust all parameters by heuristic/manually
   1,000-10,000 computers Cannot exchange all data between machines, Distributed optimization, multicore
   Large networks Nontrivial parameter dependence structure

   (d) GD + Distributed Implementation

   (e) Newton's method, Constraints - nontriv to parallelize

   (f) Bundle Methods (max-linear lower approximation, can parallelize certain regularized ERM methods)

   (g) Online methods
      i. Perceptron: Convergence Theorem
      If there exists some w,b with unit length and margin, then the perceptron converges to a linear separator
      Turns out, = Stochastic gradient descent on hinge loss [wow]
      ii. SGD

   (h) Parallel distributed variants
   Delayed Updates
   Multiple machines
   Idiot proof simple algorithm
      i. Perform stochastic gradient on each computer for a random subset of the data (drawn with replacement)
      ii. Average parameters "regularization limits parallelization"

   (i) Integers IP - "Totally unimodular"

(j) Submodular maximization eg - web search

(k) Applications
Feature selection
Active learning and experimental design
Disease spread detection in networks
Document summarization
Learning graphical models
Extensions to
Weighted item sets
Decision trees
[Feature sel: is submod more or less the same as RIP?]

5. GLM Kernel trick
   Write algorithm in terms of inner products
   Replace inn prod with k(x,x')
6. SVM
   Heuristic view
   Risk minimization view
   Find function f minimizing classification error
   Compute empirical average
   Minimization is nonconvex
   Overfitting as we minimize empirical error
   Compute convex upper bound on the loss
   Add regularization for capacity control
7. Novelty Detection
   Network Intrusion Detection Detect whether someone is trying to hack the network, downloading tons of MP3s, or doing anything else un- usual on the network.
   Jet Engine Failure Detection You cant destroy jet engines just to see how they fail.
   Database Cleaning We want to find out whether someone stored bogus in- formation in a database (typos, etc.), mislabelled digits, ugly digits, bad photographs in an electronic album.
   Fraud Detection Credit Cards, Telephone Bills, Medical Records
   Self calibrating alarm devices Car alarms (adjusts itself to where the car is parked), home alarm (furniture, temperature, windows, etc.)
   Novelty Detection via Density Estimation small density
   Structured Estimation (preview)
   PCA: Good approximation by low-rank model
8. Federated learning

6.8 AutoML

1. Reference: http://www.automl.org/book/
   Meta-data sharing: OpenML.org
2. What is AutoML?
   Traditionally, the term has been used to describe automated methods for model selection and/or hyperparameter optimization.

People often feel like they are just guessing as they test out different hyperparameters for a model, and automating the process could make this part of the machine learning pipeline easier, as well as speeding things up even for experienced machine learning practitioners.

3. Several levels: Hyperparameter Tuning, Architecture/Pipeline Search, Meta/Transfer Learning, MultiTask Learning
4. Neural Architecture Search
   (a) A search strategy selects an architecture A from a predefined search space $\mathcal{A}$. The architecture is passed to a performance estimation strategy, which returns the estimated performance of A to the search strategy. Figure 103

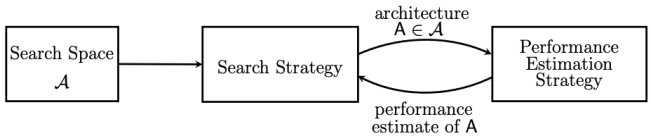


Figure 3.1: Abstract illustration of Neural Architecture Search methods. A search strategy selects an architecture A from a predefined search space $\mathcal{A}$. The architecture is passed to a performance estimation strategy, which returns the estimated performance of A to the search strategy.

Figure 103: Neural Architecture Search

Performance estimation strategy:
Input: Architecture $A = A(W)$. Output: Performance $R(A) = \min_W R(A, W)$. Say by doing a usual training with test evaluation. Fix A, minimize over W.
Search strategy:
Input: Historical performance of architectures $R(A_i)$. Output: Optimal architecture A. $\arg\min_{A \in \mathcal{A}} R(A)$.

   (b) Basic Neural Architecture Search Spaces. Figure 104

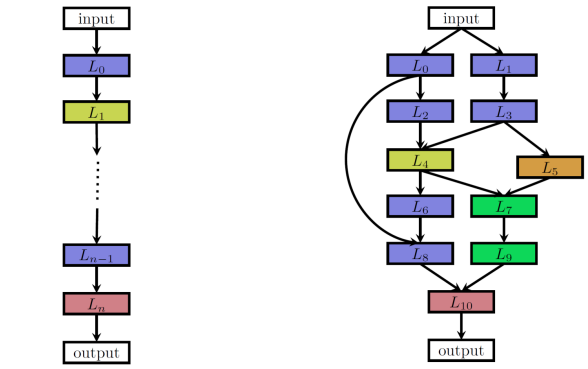


Figure 104: Basic Neural Architecture Search Spaces

"Our work is based on the observation that the structure and connectivity of a neural network can be typically specified by a variable-length string. It is therefore possible to use a recurrent network the controller to generate such string. Training the network specified by the string the child network on the real data will result in an accuracy on a validation set. Using this accuracy as the reward signal, we can compute the policy gradient to update the controller."
Policy: $\pi(A|W)$. Actions A - architectures. Weights W - weights of NN. NN implementing the policy: RNN.

(c) Cell Search Spaces. Figure 105

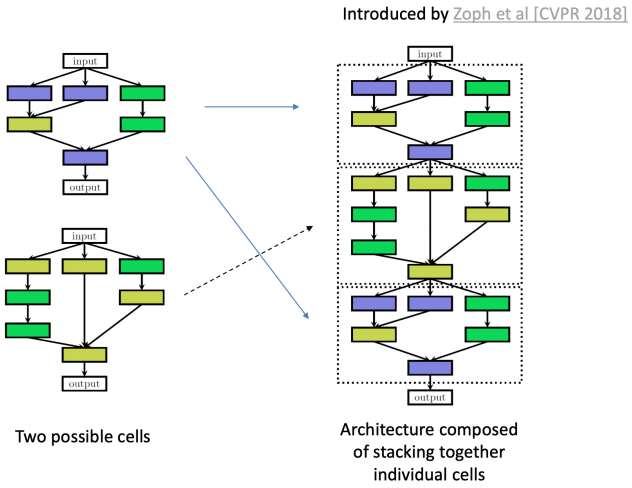


Figure 105: Cell Search Spaces.

(d) NAS as Hyperparameter Optimization. Figure 106

(e) The difficulty of the optimization:

i. non-continuous

ii. relatively high-dimensional

(f) Search Strategy

i. Bayesian Optimization. Early successes since 2013, leading to some SOTA in vision, and the first automatically-tuned neural networks to win competition datasets against human experts.

ii. Evolutionary method

iii. Reinforcement Learning. NAS became a mainstream research topic in ML community

iv. Gradient-based methods

5. RL

(a) Figure 107. Controller: Figure 108.

(b) Accelerate Training with Parallelism and Asynchronous Updates
"As training a child network can take hours, we use distributed training and asynchronous parameter updates in order to speed up the learning process of the controller (Dean et al., 2012). We use a parameter-server scheme where we have a parameter

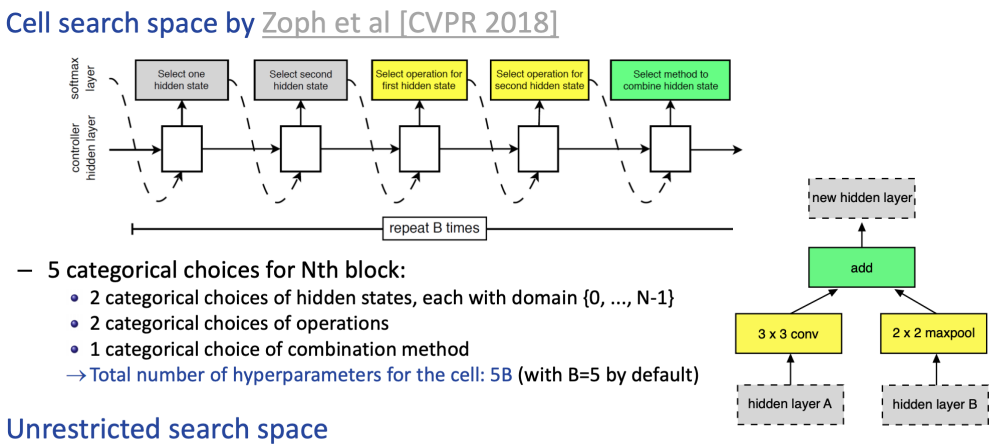


Figure 106: NAS as Hyperparameter Optimization

server of S shards, that store the shared parameters for K controller replicas. Each controller replica samples m different child architectures that are trained in parallel. The controller then collects gradients according to the results of that minibatch of m architectures at convergence and sends them to the parameter server in order to update the weights across all controller replicas."

(c) Evolution: Figure 109. Comparison: Figure 110.

(d) Making search more continuous. One-shot Architecture (Efficient Neural Architecture Search via Parameter Sharing, Pham et al, 2018):
Treats all architectures as different subgraphs of a supergraph (the one-shot model) and shares weights between architectures that have edges of this supergraph in common.
Only the weights of a single one-shot model need to be trained, and architectures can then be evaluated without any separate training by inheriting trained weights from the one-shot model.

(e) Figure 111.

(f) Details: Controller network is an LSTM with 100 hidden units. The LSTM samples decisions via softmax classifiers. Two sets of learnable parameters:

i. the parameters of the controller LSTM,

ii. the shared parameters of the child models,

Training algorithm:

i. Fix the controllers policy and perform stochastic gradient descent to minimize the expected loss function.

ii. Fix and update the policy parameters, aiming to maximize the expected reward.

6. DARTS. (DARTS: DIFFERENTIABLE ARCHITECTURE SEARCH, Liu et al, 2019) Figure 112.

- NAS with Reinforcement Learning [Zoph & Le, ICLR 2017]
  - State-of-the-art results for CIFAR-10, Penn Treebank
  - Large computational demands
    - **800 GPUs for 3-4 weeks, 12.800 architectures evaluated**

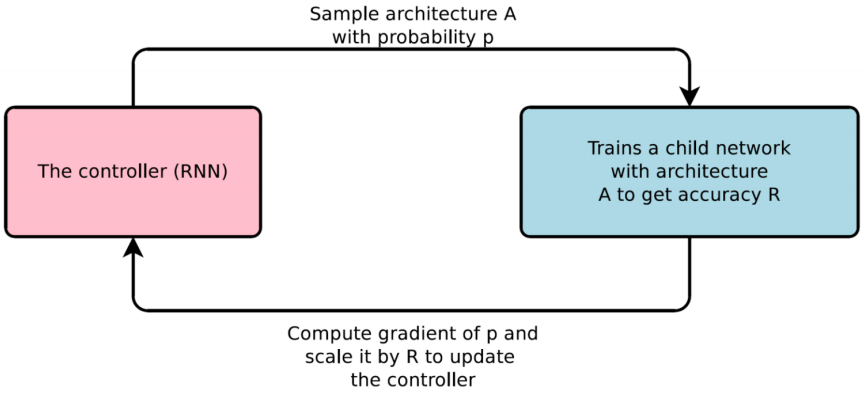


Figure 107: NAS with RL.

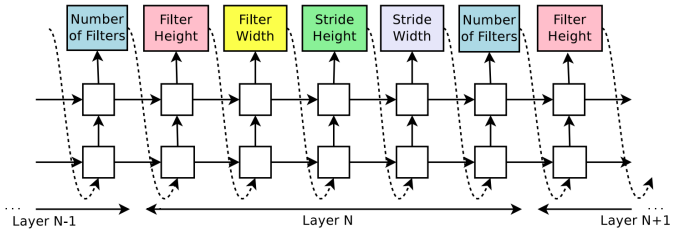


Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

Figure 108: Controller.

Relax the categorical choice of a particular operation as a softmax over all possible operations:

(a) The task of architecture search reduces to learning a set of continuous variables. "Following Zoph et al. (2018); Real et al. (2018); Liu et al. (2018a;b), we search for a computation cell as the building block of the final architecture. The learned cell could either be stacked to form a convolutional network or recursively connected to

- Neuroevolution (already since the 1990s)
  - Typically optimized both architecture and weights with evolutionary methods
    [e.g., Angeline et al, 1994; Stanley and Miikkulainen, 2002]
  - Mutation steps, such as adding, changing or removing a layer
    [Real et al, ICML 2017; Miikkulainen et al, arXiv 2017]

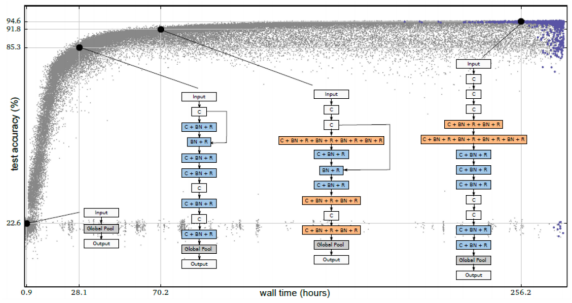


Figure 109: Evolution.

- Standard evolutionary algorithm [Real et al, AAAI 2019]
  - But oldest solutions are dropped from the population (even the best)
- State-of-the-art results (CIFAR-10, ImageNet)
  - Fixed-length cell search space

Comparison of evolution, RL and random search

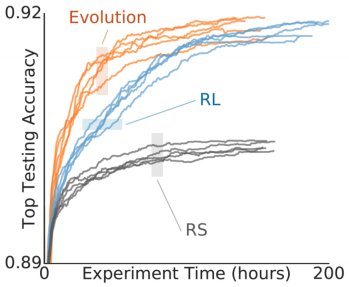


Figure 110: Comparison: Evolution, RL.

form a recurrent network."
Node $x^{(i)}$: latent representation (e.g. a feature map in convolutional networks).
Directed edge: Operation $o^{(i,j)}$ transforming input.
"Each intermediate node is computed based on all of its predecessors:"

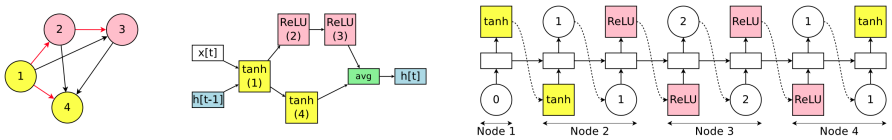


Figure 1. An example of a recurrent cell in our search space with 4 computational nodes. *Left:* The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. *Middle:* The recurrent cell. *Right:* The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output.

Figure 111: One-shot NAS.

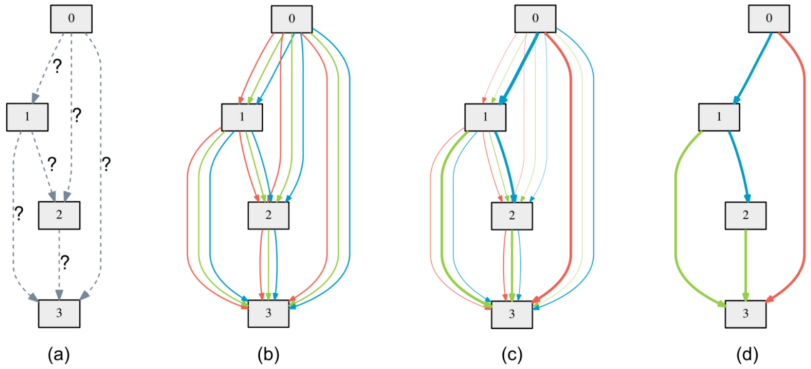


Figure 1: An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

Figure 112: DARTS

$$x^{(j)} = \sum_{i<j} o^{(i,j)}[x^{(i)}]$$

"Let O be a set of candidate operations (e.g., convolution, max pooling, zero). To make the search space continuous, we relax the categorical choice of a particular operation to a softmax over all possible operations:"

$$\bar{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x)$$

(b) A discrete architecture is obtained by replacing each mix operation with the most likely operation.

(c) "After relaxation, our goal is to jointly learn the architecture α and the weights w within all the mixed operations (e.g. weights of the convolution filters).

(d) Bilevel optimization problem (Anandalingam & Friesz, 1992; Colson et al., 2007) with α as the upper-level variable and w as the lower-level variable:

$$\min_{\alpha} \quad L_{val}(w^*(\alpha), \alpha)$$
$$s.t. \quad w^*(\alpha) = \arg\min_{w} L_{train}(w, \alpha)$$

(e) APPROXIMATE ARCHITECTURE GRADIENT

$$\nabla_\alpha L_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha L_{val}(w - \xi \nabla_w L_{train}(w, \alpha), \alpha)$$

"Related techniques have been used in meta-learning for model transfer (Finn et al., 2017), gradient- based hyperparameter tuning (Luketina et al., 2016) and unrolled generative adversarial networks (Metz et al., 2017)."
Final steps: Chain rule leads to Hessian, approximate by finite difference.

(f) Evaluation: CIFAR-10
"A large network of 20 cells is trained for 600 epochs with batch size 96." etc: cutout (DeVries & Taylor, 2017), path dropout of probability 0.2 and auxiliary towers with weight 0.4. The training takes 1.5 days on a single GPU with our implementation in PyTorch (Paszke et al., 2017).

7. Meta-Learning

(a) Meta-data: describes prior learning tasks and previously learned models.
Exact algorithm configurations: Hyperparameter settings. Pipeline compositions. Network architectures
Model evaluations: Accuracy.. Training time. Learned model parameters.
Meta-features: measurable properties of the task

(b) Learn a (base-) learning algorithm
Learn from prior meta-data, to extract and transfer knowledge that guides the search for optimal models for new tasks.

(c) Three approaches, for increasingly similar tasks
   i. Transfer prior knowledge about what works
   ii. Reason about model performance across tasks
   iii. Start from models trained earlier on similar tasks

(d) Transfer prior knowledge:
   i. Top-K recommendation:
   ii. Model hyperparameters λ_i; performance on tasks P_{ij}
   iii. Build global ranking, recommend the top-K. Requires fixed selection of candidate configurations. Warm start.
   iv. What if prior configurations are not optimal? Fit per-task estimator of performance, to GD [Reduce to supervised learning]
   v. Configuration space design: Functional ANOVA, tunability, pruning
   vi. Configuration Transfer:
   Evaluate recommended configurations on new task, yielding new evidence P_{in}. If evaluations are similar to P_{ij}, then tasks are similar.
   vii. Bayesian optimization: learns how to learn within a single task (short-term memory). Surrogate model: probabilistic regression model of configuration performance.
   Extend between models: weight by task similarity.
   viii. Surrogate models. Learn model $s_j(\theta_i) = P_{ij}$

ix. Warm-Started Multi-task Learning: learn a joint task representation using the performance measures

(e) Learning from task properties
Warm-starting from similar tasks
Meta-models
Pipelines

(f) Learning from trained models
Transfer learning
Learning to learn by GD: Our parameters probably don't do backprop. Replace it with

i. Parametric rule to update weights

ii. NN to learn weight updates

Few-shot learning
Model-agnostic meta-learning
Meta-RL: state-action-reward: abstract state-choose policy-performance

6.9 Biological plausibility

1. Todo.
2. Hebbian learning:
   "The general idea is that any two cells or systems of cells that are repeatedly active at the same time will tend to become 'associated' so that activity in one facilitates activity in the other."
3. Let $a_i[t]$ be the activation of the i-th neuron at time t. Let $W_{ij}[t]$ be the weight connecting the two neurons. Update

$$W_{ij}[t+1] = W_{ij}[t] + \eta a_i[t] a_j[t]$$

4. Questions: Does that work?
5. For unsupervised learning, a small change leads to Oja's rule http://www.scholarpedia.org/article/Oja_learning_rule, which ends up being equivalent to SGD for computing PCA
6. Recent work from Uber integrating NNs and Hebbian learning at ICLR 2019.

6.10 Accessibility

1. Motivated to read about this because of Penn AI. Figure 113.
2. "The components of PennAI include a
   human engine (i.e., the user);
   a user-friendly interface for interacting with the AI;
   a machine learning engine for data mining;
   a controller engine for launching jobs and keeping track of analytical results;
   a graph database for storing data and results (i.e., the memory);
   an AI engine for monitoring results and automatically launching or recommending new analyses;
   and a visualization engine to displaying results and analytical knowledge.

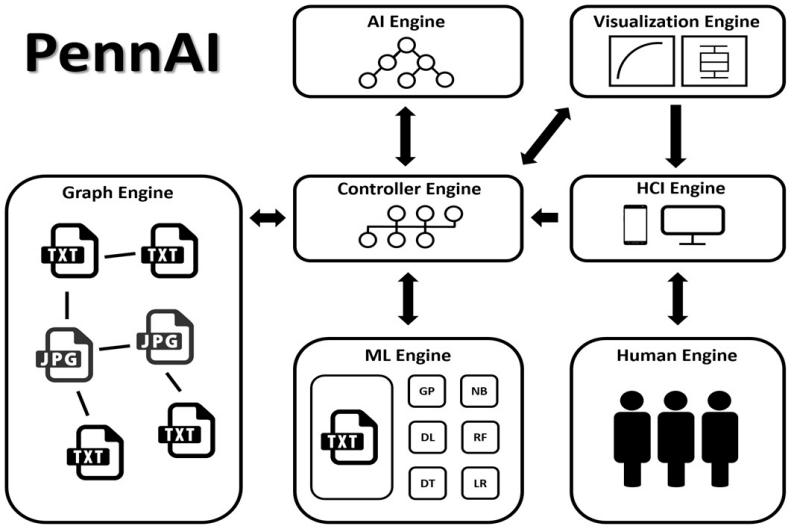


Figure 113: Penn AI

This AI system provides a comprehensive set of integrated components for automated machine learning (AutoML), thus providing a data science assistant for generating useful results from large and complex data problems. More details can be found in our PennAI publications."

3. Market: Amazon for ML. Can "buy" and rate models/tools/datasets. Can give reviews.
   Different from other platforms: Kaggle, StackOverflow, CRAN, papers (don't have review information)
   Concern: how to make the market as large as possible.

6.11 Explainability (and interpretability)

1. This is very important in many areas, such as legal, finance, marketing
   locally linear, Shapley value
   locally linear one of the most interesting papers from ICLR 2019
2. Class model visualization ("Deep inside convolutional networks: Visualising image classification models and saliency maps", Simonyan, Vedaldi, Zisserman, 2013)
   Let $S_c(I)$ be the unnormalized score of the class c, computed by the classification layer of the ConvNet for an image I. We would like to find an L_2 - regularised image, such that the score is high:

$$\arg\max_I S_c(I) - \lambda \|I\|_2^2$$

3. Saliency maps ("Deep inside convolutional networks: Visualising image classification models and saliency maps", Simonyan, Vedaldi, Zisserman, 2013)
   Image specific class saliency maps.
   Model $x \to f(x; W) := f(x)$.
   Final class is constructed as a softmax of f.

Figure 114: Saliency maps

Define saliency map for class i as starting with the partial derivatives

$$S_i = \frac{\partial f(x)}{\partial x}_i$$

Then one can take element-wise absolute values:

$$M_i = |\frac{\partial f(x)}{\partial x}_i|$$

If there are multiple channels, we can also take the max over all channels.

Has applications to "Weakly Supervised Object Localisation": "Given an image and the corresponding class saliency map, we compute the object segmentation mask using the GraphCut colour segmentation. The use of the colour segmentation is motivated by the fact that the saliency map might capture only the most discriminative part of an object, so saliency thresholding might not be able to highlight the whole object"

4. Insight into the training: Visualizing the loss surface
5. A Microscope into Neural Network Training by Measuring Per-parameter Learning (Uber AI) (Lan, Liu, Zhou, Yosinski)

   Propose "Impact on Loss" (IOL) to measure how much each parameter increases or decreases the loss at every iteration.

$$L(\theta_{t+1}) - L(\theta_t) \approx \langle \nabla_\theta L(\theta_t), \theta_{t+1} - \theta_t \rangle = \sum_{i=0}^{K-1} (\nabla_\theta L(\theta_t))^{(i)} (\theta_{t+1}^{(i)} - \theta_t^{(i)})$$

   They use parameter updates based on the minibatch but use the entire training set to calculate the gradient.

   Some analysis.

   The IOL is exactly the first term in the Taylor series approximation of the loss, and so one can probably say a lot about it.

Suppose that the update rule is GD (non-stochastic).

$$\theta_{t+1} = \theta_t - \eta_t \nabla_\theta L(\theta_t)$$

Then the IOL is

$$I_t = \langle \nabla_\theta L(\theta_t), \theta_{t+1} - \theta_t \rangle = -\eta_t \|\nabla_\theta L(\theta_t)\|^2 = -\eta_t \sum_{i=0}^{K-1} \nabla_\theta L(\theta_t)^{(i),2} \leq 0$$

Thus, in this case the IOL of each coordinate is negative, so that each coordinate "learns". Note that this holds even if the loss is non-convex. In particular, for a non-convex loss, the change between the loss values at two parameters may be positive. In that case the IOL will not be an accurate approximation.

The above shows that the IOL can only be positive due to the use of a different training algorithm: either the stochasticity or adaptive learning rule such as momentum, Adam etc. In some sense, it can be viewed as measuring the deviation from the ideal case in which we can compute the full gradient. It also shows that using the full gradient to compute the IOL may be too expensive, as for that cost we could in fact take the full gradient step.

Suppose that we use the SGD over a minibatch with indices in S_t:

$$\theta_{t+1} = \theta_t - \tilde{\eta}_t \nabla_\theta L(\theta_t; S_t)$$

Here $\tilde{\eta}_t = n/|S_t|\eta_t$ is the learning rate, with a scaling to ensure normalization.

Then the IOL is, assuming we calculate the gradient part on the entire training set:

$$I_t = \langle \nabla_\theta L(\theta_t), \theta_{t+1} - \theta_t \rangle = -\tilde{\eta}_t \langle \nabla_\theta L(\theta_t), \nabla_\theta L(\theta_t; S_t) \rangle$$

This shows that the IOL is a sort of "angle" between the stochastic and real gradient. Such terms arise often in the analysis of SGD (see e.g., the review paper by Bottou, Curtis, Nocedal, "Optimization methods for large-scale ML"). In particular, the convergence properties of SGD depend on the mean and variance of the stochastic gradient (See e.g., Thm 4.9 in that work). In those analyses, one must show that the IOL is larger than the remainder of the Taylor approximation, so that the loss indeed decreases.

6.12 Model compression

1. Model compression is important and useful in practice, as it saves resources
2. Sparsification - intuitive, but not always practical, because want to implement it fast in hardware
3. Structured convolutions: tensor factorizations, eg PCA. Important problem, how to choose the rank
4. Quantization:
5. Lottery ticket hypothesis. FINDING SPARSE, TRAINABLE NEURAL NETWORKS, Frankle, Carbin, 2019

   NNs are overparametrized. Can we prune them?

   "Techniques for eliminating unnecessary weights from neural networks (pruning) (LeCun et al., 1990; Hassibi & Stork, 1993; Han et al., 2015; Li et al., 2016) can reduce parameter-counts by more than 90% without harming accuracy. Doing so decreases the size (Han

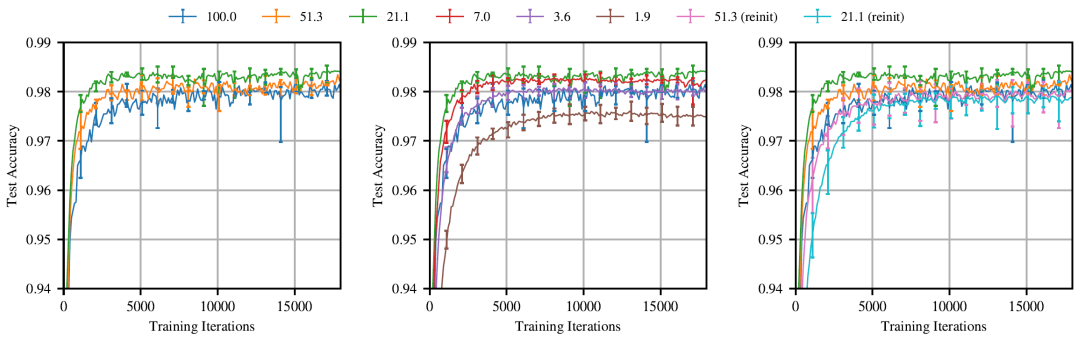


Figure 3: Test accuracy on Lenet (iterative pruning) as training proceeds. Each curve is the average of five trials. Labels are P_m—the fraction of weights remaining in the network after pruning. Error bars are the minimum and maximum of any trial.

Figure 115: Lottery ticket hypothesis

et al., 2015; Hinton et al., 2015) or energy consumption (Yang et al., 2017; Molchanov et al., 2016; Luo et al., 2017) of the trained networks, making inference more efficient."

"However, if a network can be reduced in size, why do we not train this smaller architecture instead in the interest of making training more efficient as well? Contemporary experience is that the architectures uncovered by pruning are harder to train from the start, reaching lower accuracy than the original networks"

"In this paper, we show that there consistently exist smaller subnetworks that train from the start and learn at least as fast as their larger counterparts while reaching similar test accuracy. Based on these results, we state:

The Lottery Ticket Hypothesis. A randomly-initialized, dense neural network contains a subnetwork that is initialized such that when trained in isolation it can match the test accuracy of the original network after training for at most the same number of iterations."

Algorithm:

(a) Randomly initialize a neural network $f(x; W_0)$ (where $W_0 \sim D$).
(b) Train the network for j iterations, arriving at parameters W_j.
(c) Prune $p\%$ of the parameters in W_j, creating a mask m.
(d) Reset the remaining parameters to their values in W_0, creating the winning ticket $f(x; m \odot W_0)$

"we extend our hypothesis into an untested conjecture that SGD seeks out and trains a subset of well-initialized weights"

6.13 Others

6.13.1 List

1. Bayesian NNs
   Uncertainty quantification
   A simple Bayesian baseline for deep learning
   Osband bootstrap nn

2. Learning from limited data
   Semi-supervised learning: using unlabeled data, clustering assumption (Bengio, Ermon)
   Active learning
3. Bayesian optimization
4. Domain adaptation: Nowadays you often want to adapt to new environments, much more so than in the past.
   examples: sim2real.
5. Neuroscience:
   Intelligence is the biggest problem. Not just engineering, also science.
   Poggio hmax. A classic deep architecture/model for human vision.
6. Research frontiers:
   How to integrate human knowledge. Eg syntactic trees in linguistics. Can predict better than humans, but can't explain why.

6.13.2 Privacy

1. Motivated to read about this because of discussions with researchers in Penn Biostats.
2. "No Peek: A Survey of private distributed deep learning"

6.13.3 Applications

1. Industry has more resources: data, compute, manpower. For problems of immediate business value, they can leverage these resources, and it becomes hard to compete with.
2. In academia, we can leverage domain experts. For instance: science (physics, chemistry, neuroscience).
   Industry is less interested in these, because they do not bring immediate value to the customers. Also, they may have difficulty getting some experiments done (such as monkeys in their labs). Their shareholders may be unhappy.

7 Theory

7.1 Why do we need theory?

- Theory is key for mankind: "If you understand it, you can use it."
  - Physics: Newton, Maxwell → Vehicles, Devices
  - Chemistry: Molecules → Drugs
  - Computer science: Algorithms → Software
- Without theory, we are left with trial and error. Need to understand how they make decisions
- National AI RD strategic plan includes: What are the theoretical limitations of AI?
- But, there are also problems for which we have no theory, and yet we rely on them every day [?]
- A successful theory...
  - Explains and predicts, compresses observations
  - Suggests improvements, better architectures

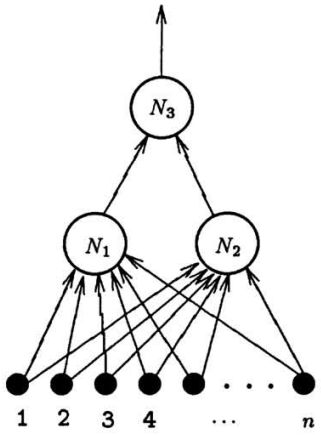


Figure 116: 3-Node Neural Network that is NP-Complete to train. Blum and Rivest

  - Elegant
  - Widely used
- So, far not very successful
  - Unrealistic: population-based, one-layer, linearized, Gaussian RF, synthetic (not trained), no computation
  - Imprecise
  - Not used much
- Fundamental differences between practitioners and theorists. Only care about improving, not about why.
- Theorists need to do experiments.
- At a fundamental level, people ask themselves: Will current math be enough to understand them?

  High dimensional places are the new frontier

7.2 Computational complexity and learning

1. Judd (1987) shows the following problem to be *NP-complete*:

   "Given a neural network and training data, are there edge weights so that the network fits the data perfectly?"

   Blum and Rivest (1989) extend his result by showing that it is NP-complete to train a specific very simple network, having only two hidden nodes and a regular interconnection pattern. Their results state that given a network in the class considered, for any algorithm there will be some training data such that the algorithm will perform poorly.

   Given: A set of $O(n)$ training examples on n inputs, x_i, y_i, with $y_i = \pm 1$.

   Question: Do there exist linear threshold functions f_1, f_2, f_3 of the form

   $$f_i(x) = \text{sign}(a_i^\top x + b_i)$$

   for nodes N_1, N_2, N_3 such that the network of figure 1 fits the training data?

Theorem 7.1 (Blum and Rivest (1989)). *Training a 3-node neural network is NP-complete.*

Proof idea:

We may reduce our question to the following: given $O(n)$ points in $\{0,1\}^n$, each point labeled '+' or '-', does there exist either

(a) a single plane that separates the '+' points from the '-' points, or

(b) two planes that partition the points so that either one quadrant contains all and only '+' points or one quadrant contains all and only '-' points.

Reduce SET-SPLITTING to this problem.
Given: A finite set S and a collection C of subsets C_i of S.
Question: Do there exist disjoint sets S_1, S_2 such that $S_1 \cup S_2 = S$ and for all i, C_i are not subsets of either S_i

2. "Learning Two Layer Rectified Neural Networks in Polynomial Time", Bakshi, Jayaram, Woodruff

Two-layer neural networks from d to m dimesnions, with a single hidden layer and k non-linear activation units $f()$, where $f(x) = max(x, 0)$ is the ReLU activation function.

It is known that stochastic gradient descent cannot converge to the ground truth parameters when f is ReLU and V^* is orthonormal, even if we have access to an infinite number of samples [LSSS14]. This is consistent with empirical observations and theory, which states that over-parameterization is crucial to train neural networks successfully [Har14, SC16].

[LSSS14] Roi Livni, Shai Shalev-Shwartz, and Ohad Shamir. On the computational efficiency of training neural networks. In Advances in Neural Information Processing Systems, pages 855863, 2014.

"One of our primary techniques throughout this work is the leveraging of combinatorial aspects of the ReLU function. For a row $f(V^*X)_{i,*}$, we define a sign pattern of this row to simply be the subset of positive entries of the row. Thus, a sign pattern of a vector in $\mathbb{R}^n$ is simply given by the orthant of $\mathbb{R}^n$ in which it lies. We first prove an upper bound of $O(n^k)$ on the number of orthants which intersect with an arbitrary k-dimensional subspace of $\mathbb{R}^n$. Next, we show how to enumerate these sign patterns in time $n^k + O(1)$."

Theorem 1. Given $A \in \mathbb{R}^{m\times n}$, $X \in R^{d\times n}$, such that $A = U^* f(V^* X)$ and A is rank k, there is an algorithm that finds $U^* \in \mathbb{R}^{m\times k}, V^* \in \mathbb{R}^{k\times d}$ such that $A = U^* f(V^* X)$ and runs in time $poly(n, m, d)min(n^{O(k)}, 2^n)$.

"We demonstrate that an $O(n^k)$ time algorithm is no longer possible without this assumption by proving the NP-hardness of the realizable learning problem when $rank(A) < k$, which holds even for k as small as 2.

Theorem 2. For a fixed $\alpha \in \mathbb{R}^{m\times k}$, $X \in R^{d\times n}$, $A \in \mathbb{R}^{m\times n}$, the problem of deciding whether there exists a solution $V^* \in \mathbb{R}^{k\times d}$ to $\alpha f(VX) = A$ is NP-hard even for $k = 2$. Furthermore, for the case for $k = 2$, the problem is still NP-hard when α is allowed to be a variable.

Theorem 7. Given $A \in \mathbb{R}^{m\times n}$, $X \in R^{d\times n}$, such that $A = U^* f(V^* X)$ and A is rank k, where U^*, V^* are both rank k, and such that the columns of X are mean 0 i.i.d. Gaussian. Then if $n = \Omega(poly(d, m, k(U^*), k(V^*)))$, then there is a poly(n)-time algorithm which recovers U^*, V^* exactly up to a permutation of the rows with high probability.

"To the best of our knowledge, this is the first algorithm which learns the weights matrices of a two-layer neural network with ReLU activation exactly in the noiseless case and with Gaussian in- puts. Our algorithm first obtains good approximations to the weights, and

concludes by solving a system of judiciously chosen linear equations, which we solve using Gaussian elimination. Therefore, we obtain exact solutions in polynomial time, without needing to deal with convergence guarantees of continuous optimization primitives."

Idea:

A is rank k. Thus, there is a subset V in $R^{k \times n}$ of k rows of A which span all the rows of A.

So there is an invertible $k \times k$ matrix W such that

$$WV' = f(V^*X)$$

$$sign(V^*X) = sign(f(V^*X)) = sign(WV')$$

thus it suffices to return a set of sign patterns which contains $sign(WV')$.

Therefore, consider any fixed sign pattern $S \subset [n]$, and fix a row $j \in [k]$, and consider the following feasibility linear program in the variables w_j (w_j are k-dimensional rows of W)

$$(w_j V')_i \geq 1$$

for all $i \in sign(S)$

$$(w_j V')_i \leq 0$$

for all $i \notin sign(S)$.

(Positive inner product with all columns of V' in S, and negative with all columns in the remainder)

Now the LP has k variables and n constraints, and thus a solution is obtained by choosing the w_j that makes a subset of k linearly independent constraints tight.

Observe in any such LP of the above form, there are at most 2n possible constraints that can ever occur.

Since there are at most $\binom{2n}{k} = O(n^k)$ such possible choices, it follows that there are at most $O(min(n^O(k), 2^n))$ realizable sign patterns, and these can be enumerated in the same order of time by simply checking the sign pattern which results from the solution (if one exists) to $w_j S' = b'$ taken over all subsets S', b' of constraints of size k.

[Take a set of columns of V' corresponding to S', and solve feasibility problem. If it has solution, add S' to constraint set]

Note that S does not have to have size k. In fact this is actually what was confusing to me. S can have an arbitrary size, and thus in principle there are at most 2^n choices for S. However, there are only $2n$ possible *constraints*, out of which k have to be tight. So this limits the number of distinct solutions. And each solution translates into realizable sign patterns.

Next: "Given access to the set of candidate sign patterns, define an iterative feasibility linear program, that at each iteration finds a vector y_i in the row span of X, and such that $f(y_1), f(y_2), ..., f(y_i)$ are all linearly independent and in the row span of A."

Then, iterate over all patterns S, and for all obtained Y, solve linear systems $A = U f_S(Y)$, $Y = VX$ for U, V

7.3 Approximation theory

There are many results showing that neural networks of various forms have "good" approximation properties.

1. Cybenko 1989- NNs are "universal approximators"
   Telgarsky 16
2. Computational perspectice. Sipser, 86, Hastad, 87
3. **Theorem 7.2** (Barron, 1993)**.** *Let f be a function $f : [-1, 1]^p \to \mathbb{R}$. It can be approximated by linear combinations of linearly scaled sigmoids*

   $$\phi(x) = (1 - e^{-x})/(1 + e^{-x}),$$

   of the form

   $$g(x) = \sum_{k=1}^{n} c_k \phi(a_k^\top x + b_k) + c_0$$

   with accuracy

   $$\int |f(x) - g(x)|^2 dx \leq \frac{C_f^2}{n}.$$

   Here

   $$C_f = \int |x|_1 |\hat{f}(x)| dx < \infty,$$

   where $\hat{f}(x)$ is the Fourier transform of f,

   $$f(y) = \int \exp(2\pi i x^\top y) \hat{f}(x) dx.$$

   Also $|x|_1 = \sum_j |x_j|$.
4. **Theorem 7.3** (Mhaskar, Poggio, Liao'16)**.** *Shallow 1-hidden layer networks with non-polynomial nonlinearity are universal approximators. Deep networks with non-linear non-linearity are universal approximators.*

   $$\phi(x) = \sum_i c_i [w_i^\top x]_+$$

   Curse of dimensionality. One way to avoid it - increase smoothness [Barron]
   Their approach: "Hierarchical local compositionality"

   Theorem 7.4 (Mhaskar, Poggio, Liao'16)**.** *Suppose that a function of d variables is hierarchically locally compositional. Both shallow and deep networks can approximate f equally well. The number of parameters of the shallow network depends exponentially on d as $O(\varepsilon^{-d})$ whereas for a deep network it is $O(d\varepsilon^{-2})$.*

   Proof outline: We want to propagate the approximation forward layer by layer

   $$f(f_1, f_2) - g(g_1, g_2) = f(f_1, f_2) - f(g_1, g_2) + f(g_1, gf_2) - g(g_1, g_2)$$

   $$|f(f_1, f_2) - g(g_1, g_2)| \leq |f(f_1, f_2) - f(g_1, g_2)| + |f(g_1, g_2) - g(g_1, g_2)|$$

First term: $|f_i - g_i| \leq \varepsilon$ by induction step, and because f is Lipschitz.
Second term: $|f - g| \leq \varepsilon$ is small by one-step approximation
Approxiation accuracy degrades by constant multiple
After n layers, we get error $c^{\log n}\varepsilon_0 < \varepsilon$
But $\varepsilon_0 = d^{-r/2}$, finishing the proof [??]

7.4 Optimization

1. Simplest method (SGD) basically "works". How? Why is it so easy to optimize? Can it be made better?
2. Poggio: Overparametrization
   (a) "SGD should converge most of the time to global minima"
   He: semi-formal
   (b) Approximate ReLU with high degree poly. Has same behavior
   (c) Bezout theorem for poly networks: $f(x_i) = y_i$, $i = 1, \ldots, N$ The set of poly equations above with $k = deg(f)$ has a number of distinct zeros (counting points at infinity, using projective space, assigning an appropriate multiplicity to each intersection point, and excluding degenerate cases), equal to $Z = k^N$, the product of the degrees of each of the equations
   - many global minima, solutions of $f(x_i) = y_i$ - big space
   eg linear model $X^\top X\beta = X^\top Y$ [but they are linearly dependent??]
   [Why are nonzero minima not degenerate, isolated?]
   (d) Critical values $\nabla L(w) = 0$ - W equations in W unknowns, so it's a much smaller set. So: global minima are degenerate. Other critical points of the gradient are isolated, generically. [??]
   (e) Langevin equation (add "white noise" to GD or SGD)

$$\frac{df}{dt} = -\gamma_t \nabla V(f(t)) + dB(t)$$

   SGD

$$f_{t+1} = f_t - \gamma_t \nabla V(f_t, z_t)$$

   "It is interesting that the following equation, labeled SGDL, and studied by several authors, including [7], seem to work as well as or better than the usual repeat SGD used to train deep networks," SGDL

$$f_{t+1} = f_t - \gamma_t \nabla V(f_t, z_t) + \gamma_t' W_t$$

   With the Boltzmann equation as the asymptotic "solution"

$$p(f) \sim \frac{1}{Z} = \exp^{-V(x)/T}$$

   (f) Concentration because of high-dimensionality - "if you have minima that have a larger volume than others, most of the volume will concentrate on them, especially in higher dimensions"
   "SGDL finds with very high probability large volume, zero minimizers; empirically SGD behaves in a similar way"

(g) We consider a situation in which the expected cost function can have, possibly multiple, global minima. As argued by [8] there are two ways to prove convergence of SGD. The first method consists of partitioning the parameter space into several attraction basins, assume that after a few iterations the algorithm confines the parameters in a single attraction basin, and proceed as in the convex case. A simpler method, instead of proving that the function f converges, proves that the cost function and its gradient converge.

7.5 Generalization

1. Why does a net generalize, if the number of parameters is so much larger than the training data size?
2. Poggio:
   (a) GD optimization induces a gradient dynamical system
   (b) Exponential loss for classification: $L = \sum_n^N \exp(-y_n f(W_K, \dots, W_1; x_n))$
   Gradient dynamics: Negative Derivative of loss with respect to w_k^{ij}

$$\dot{w}_k^{ij} = -\eta \nabla_{w_k^{ij}} L(w) = \eta \sum_n^N y_n \exp(-y_n f(x_n)) \frac{\partial f(W; x_n)}{\partial w_k^{ij}}$$

   Hessian: Second derivative of loss with respect to w_k^{ij}, $w_{k'}^{ab}$ [use that $y_n^2 = 1$]

$$H_{abk'}^{ijk} = \sum_n^N \exp(-y_n f(x_n))[\frac{\partial f(W; x_n)}{\partial w_k^{ij}} \frac{\partial f(W; x_n)}{\partial w_{k'}^{ab}} - y_n \frac{\partial^2 f}{\partial w_k^{ij} \partial w_{k'}^{ab}}]$$

   Hessian is in general degenerate
   (c) Deep ReLU networks: 1-homogeneity

$$f(W_K, \dots, W_1; x) = W_k \sigma(\dots \sigma(W_1 x) \dots)$$

$$\sigma(z) = z \partial \sigma(z) / \partial(z)$$

$$f(W; x) = \prod \rho_k \tilde{f}(V; x)$$

   where

$$\rho_k v_k^{ij} = w_k^{ij}$$

   $\rho_k^2 = \sum_{ij} (w_k^{ij})^2$
   So that $\|V_k\| = 1$
   [Actually, I think that the property that they are using is simply that $|cx| = c|x|$, when $c \geq 0$. So you can sequentially normalize each layer.]
   Define $\rho = \prod \rho_k$ to be the norm of the network.

$$f = \rho \tilde{f}$$

   Let $R_N(F)$ be the empirical Rademacher complexity.

$$R_N(F) = \frac{1}{N} \mathbb{E} \sup_{f \in F} \sum_i \sigma_i f(x_i)$$

   Then

$$R_N(F) = \rho R_N(\tilde{F})$$

(d) Deep ReLU networks + magic of normalization: perfect generalization
Classical bounds [Bartlett, Foster, Telgarsky]: with probability $\geq 1 - \delta$

$$\mathbb{E}L_{01}(f) \leq L_{exp}(f) + \frac{2}{\eta}R(F) + \sqrt{\frac{\ln 1/\delta}{2N}}$$

where η is the margin. For a point x, the margin is $\eta \sim yf(x) \sim y\rho\tilde{f}(x)$. Since the complexity scales with ρ, the margin bound is optimized by effectively maximizing $\tilde{f}$ on the support vectors.
Regression: "Classical generalization bounds for regression suggest that bounding the complexity of the mini- mizer provides a bound on generalization. Ideally, the optimization algorithm should select the smallest complexity minimizers among the solutions that is, in the case of ReLU networks, the minimizers with minimum norm. An approach to achieve this goal is to add a vanishing regularization term to the loss function (the parameter goes to zero with iterations) that, under certain conditions, provides convergence to the minimum norm minimizer, independently of initial conditions. This approach goes back to Halpern fixed point theorem [33]; it is also independently suggested by other techniques such as Lagrange multipliers, normalization and margin maximization theorems [9]."
We show (see Appendix 10) that for separable data, maximizing the margin subject to unit norm constraint is equivalent to minimize the norm of f subject to a constraint on the margin. A regularized loss with an appropriately vanishing regularization parameter is a closely related optimization technique. For this reason we will refer to the solutions in all these cases as minimum norm. This view treats interpolation (in the regression case) and classification (in the margin case) in a unified way.

(e) Lagrange multiplier.
Proposition: GF (on Lagrange penalized problem) converges to max margin solutions with "unit" complexity $R_N(\tilde{F})$ [so it kind of works]

$$\dot{\rho}_k = \eta\frac{\rho}{\rho_k}\sum_n^N y_n \exp(-y_n\rho\tilde{f}(V; x_n))\tilde{f}(x_n)$$

Four almost equivalent techniques: Lagrange, coefficient normalization (related to batch norm), tangent gradient (weight norm), true gradient (natural gradient)
Easy to see that $d\rho_k^2/dt = d\rho_{k-1}^2/dt$. So the norms are automatically balanced [Lee, Hu, Du '17].
GD or SGD with weight or batch normalization implement generalization bounds For epsilon initialization gradient descent by itself has implicit regularization (similar to the linear, pseudo inverse case)
"Many of the future breakthroughs will be inspired by neuroscience. It will be hard for Google to get monkeys in their labs"

7.6 Harmonic analysis

There is a line of work started by Mallat (2012) to try to explain how CNNs extract features, using ideas from harmonic analysis to exploit the invariance properties of images.

See Figure 117 for a representation of a Deep Scattering Network.

Theorem 7.5 (Boelcskei & Wiatkowski, 2015). *Let $f_t(x) = f(x + t)$ be a translation of f.*

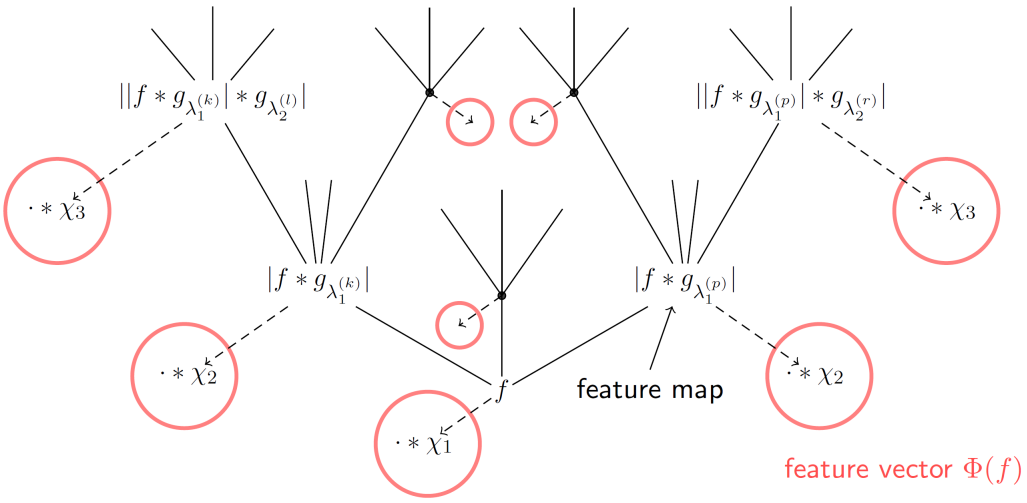


Figure 117: Deep Scattering Network, Boelcskei & Wiatkowski, 2015

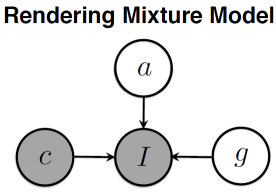


Figure 118: Rendering Mixture Model, Baraniuk, Patel, etc

The features Φ^n of the Deep Scattering Network at n-th stage are approximately translation invariant, in the sense that

$$|\Phi^n(f) - \Phi^n(f^t)| = O(|t|/c^n).$$

7.7 Probabilistic ML

There is line of work proposing probabilistic models which justify and explain neural networks. Contributors: Baraniuk, Patel, etc

See Figure 118 for a depiction of a Rendering Mixture Model.

- Object $c \sim Cat(\{\pi_c\}_{c \in C})$
- Nuisance $g \sim Cat(\{\pi_g\}_{g \in G})$
- Switch $a \sim Bern(\{q_a\}_{a \in 0,1})$
- Image $I \sim \mathcal{N}(a\mu_{cg}, I_p)$

Theorem 7.6. *MAP inference has a form similar to one layer of a NN:*

$$\hat{c} = \arg\max_c \max_g ReLU\{W_{cg}^\top I + b_{cg}\},$$

where $W_{cg} = \mu_{cg}$, $b_{cg} = -|\mu_{cg}|^2 + logit(q_1)$.

Inference in DRMM: CNN.

7.8 Information geometry

1. An extensive analysis of the Riemannian geometry of random neural networks geometry, performed by Poole et al. (2016), revealed the existence of two phases in deep networks: a chaotic (ordered) phase when the random weights have high (low) variance. In the chaotic (ordered) phase the network locally expands (con tracts) input space everywhere. In contrast, trained net works flexibly deploy both ordered and chaotic phases dif ferentially across input space; they contract space at the center of decision volumes and expand space in the vicin ity of decision boundaries.

   Ref: Poole, B., Lahiri, S., Raghu, M., Sohl-Dickstein, J., and Ganguli, S. Exponential expressivity in deep neural net works through transient chaos. In Advances in Neural Information Processing Systems (NIPS 2016), pp. 3360 3368, 2016.

7.9 Random Matrix Theory

1. Notes below from Advani & Saxe: High-dimensional dynamics of generalization error in neural networks
2. Key older ref:

   LeCun, Y., I. Kanter, and S.A. Solla (1991). Eigenvalues of covariance matrices: Application to neural-network learning. In: Physical Review Letters 66.18, p. 2396.
3. $y = X\bar{w} + \varepsilon$

   $E_t = 1/2 \cdot |y - Xw(t)|^2 = |y|^2 - 2w^\top (X^\top y) + w^\top (X^\top X)w/2$

   $dE_t/dw(t) = (X^\top X)w - X^\top y$

   gradient descent

   $\tau \cdot dw(t)/dt = X^\top y - (X^\top X)w$

   Generalization error

   $E_g(t) = \mathbb{E}_{x,\varepsilon}(y - \hat{y})^2$
4. To analyze, write

   $X = U\Lambda^{1/2}V^\top$,

   $X^\top X = V\Lambda V^\top$

   and

   $s = V^\top X^\top y$

   Let $z - V^\top w$

   $\tau \cdot dz(t)/dt = s - \Lambda z$

   This is a linear ODE that decouples over coordinates

   $\tau \cdot z_i' = s - \lambda_i z_i$

   $s = V^\top X^\top y = V^\top X^\top (X\bar{w} + \varepsilon) = V^\top X^\top (X\bar{w} + \varepsilon) = \Lambda V^\top \bar{w} + \Lambda^{1/2} U^\top \varepsilon$

   Let $V^\top \bar{w} = \bar{z}$

   Under Gaussian assumption, $\tilde{\varepsilon} = U^\top \varepsilon \sim \mathcal{N}(0, I_p \sigma_\varepsilon^2)$

   $\tau \cdot z_i' = \lambda_i(\bar{z}_i - z_i) + \lambda_i^{1/2}\tilde{\varepsilon}_i$

   Solving the ODE

   $\bar{z}_i - z_i = (\bar{z}_i - z_i(0)) \cdot \exp(-\lambda_i t/\tau) - \lambda_i^{-1/2}\tilde{\varepsilon}_i \cdot [1 - \exp(-\lambda_i t/\tau)]$

   Thus, gen error

$E_g(t) = n^{-1}\sum_i \mathbb{E}(\bar{z}_i - z_i)^2 + \sigma_\varepsilon^2 = n^{-1}\sum_i [(\sigma_w^2 + \sigma_0^2)\exp(-2\lambda_i t/\tau) + \sigma_\varepsilon^2/\lambda_i(1-\exp(-\lambda_i t/\tau))^2] + \sigma_\varepsilon^2$

σ_w - SD of true weight distrib

σ_0 - SD of initial weight distrib

σ_ε - SD of noise

5. High-dimensional asymptotic limit: take $n, p \to \infty$ such that $p/n \to \gamma > 0$

   See learning dynamics:

   2.2 Effectiveness of early stopping vs L2 regularization: early stopping comes close.

   2.3 Optimal stopping time vs SNR

   2.4 Weight norm growth with SNR

   2.5 Impact of initial weight size on generalization performance

   2.6 Training error dynamics

Another set of notes: Pennington et al: Geometry of NN loss landscapes

1. Deep learning has analogies with stat mechanics. Why random matrix theory? Randomness is used in initialization. Also, exact theory uninformative

   Surprising finding: deep Relu networks have a worse conditioning than sigmoid networks [only theory?]

2. Activations in a feedforward NN

$$Y_{i\mu}^l = f(\sum_j W_{ij}^l Y_{j\mu}^{l-1})$$

   (a) layer l

   (b) neuron i

   (c) example μ [does not vary]

3. Tools: Stieltjes transform $G(z)$

   R transform

$$1/G(z) + R(G(z)) = z$$

$$S(zG(z) - 1) = G(z)/[zG(z) - 1]$$

4. Index of a critical point - fraction of negative eigenvalues of Hessian (0 - global min, 1 - global max)

   Known:

   All minima global: deep linear net (Kawaguchi 2016 0 – surprising), non-linear net with pyramidal structure (Nguyen Hein 2017)

5. What is the distribution of critical points? (Choromanska et al 2014),

   Gaussian random fields: expression for index as a function of energy

$$\alpha \sim 42^{1/2}/3\pi[|E - E_c|/|E_c|]^{3/2}$$

   Compute Hessian, and make rm approximations, get approx index. Still 3/2 index

   Empirically ok with simulated data, but not yet with real data

7.10 Physics

1. Key ref: Why does deep and cheap learning work so well? Henry W. Lin, Max Tegmark, and David Rolnick
2. We show how the success of deep learning could depend not only on mathematics but also on physics: although well-known mathematical theorems guarantee that neural networks can approxi- mate arbitrary functions well, the class of functions of practical interest can frequently be approxi- mated through cheap learning with exponentially fewer parameters than generic ones. We explore how properties frequently encountered in physics such as symmetry, locality, compositionality, and polynomial log-probability translate into exceptionally simple neural networks. We further argue that when the statistical process generating the data is of a certain hierarchical form prevalent in physics and machine-learning, a deep neural network can be more efficient than a shallow one. We formalize these claims using information theory and discuss the relation to the renormalization group. We prove various no-flattening theorems showing when efficient linear deep networks can- not be accurately approximated by shallow ones without efficiency loss; for example, we show that n variables cannot be multiplied using fewer than 2 n neurons in a single hidden layer.
3. Parameters y, data x. Probability model $p(x|y)$
   Physics: Hamiltonian, quantifying the energy of data given the parameter.

   $$H_y(x) = -\ln p(x|y),$$

   Bayes theorem can be viewed as a softmax
4. What Hamiltonians can be approximated by feasible neural networks?
   both physics and machine learning tend to favor Hamiltonians that are polynomials

   $$H_y(x) = h + \sum_i h_i x_i + \sum_{i \le j} h_{ij} x_i x_j + \sum_{i \le j \le k} h_{ijk} x_i x_j x_k + \dots$$

   If we can accurately approximate multiplication using a small number of neurons, then we can construct a net- work efficiently approximating any polynomial by repeated multiplication and addition.
5. What Hamiltonians do we want to approximate?
   (a) Low degree: The Hamiltonians that show up in physics are not ran dom functions, but tend to be polynomials of very low order, typically of degree ranging from 2 to 4. The sim- plest example is of course the harmonic oscillator, which is described by a Hamiltonian that is quadratic in both position and momentum.
   At a fundamental level, the Hamiltonian of the stan- dard model of particle physics has d = 4. There are many approximations of this quartic Hamiltonian that are accurate in specific regimes, for example the Maxwell equations governing electromagnetism, the Navier-Stokes equations governing fluid dynamics, the Alven equa tions governing magnetohydrodynamics and various Ising models governing magnetization - all of these approxi- mations have Hamiltonians that are polynomials in the field variables, of degree d ranging from 2 to 4.
   (b) Locality: One of the deepest principles of physics is locality: that things directly affect only what is in their immediate vicinity.
   (c) Symmetry: Whenever the Hamiltonian obeys some symmetry (is invariant under some transformation), the number of independent parameters required to describe it

is further reduced. For instance, many probability distributions in both physics and machine learning are invariant under translation and rotation.
As an example, consider a vector x of air pressures y_i measured by a microphone at times $i = 1, ..., n$. Assuming that the Hamiltonian describing it has $d = 2$ reduces the number of parameters N from infinity to $(n + 1)(n + 2)/2$. Further assuming locality (nearest-neighbor couplings only) reduces this to N = 2n, after which requiring translational symmetry reduces the parameter count to N = 3.
Taken together, the constraints on locality, symmetry and polynomial order reduce the number of continuous parameters in the Hamiltonian of the standard model of physics to merely 32 [13]
[13] M. Tegmark, A. Aguirre, M. J. Rees, and F. Wilczek, Physical Review D 73, 023505 (2006)

(d) III. WHY DEEP?
A. Hierarchical processess
In neuroscience parlance, the functions f_i compress the data into forms with ever more invariance [28], containing features invariant under irrelevant transformations (for example background substitution, scaling and trans lation).
[28] M. Riesenhuber and T. Poggio, Nature neuroscience 3, 1199 (2000).
C. Sufficient statistics and hierarchies
F. No-flattening theorem
For example, when A represents the discrete Fourier transform (DFT), the fast Fourier transform (FFT) algorithm makes use of a sparse factorization of A which only contains $O(n \log n)$ non-zero matrix elements instead of the naive single-layer implementation, which contains n^2 non-zero matrix elements. As first pointed out in [46], this is an example where depth helps and, in our termi- nology, of a linear no-flattening theorem
Matrix multiplication
H. A polynomial no-flattening theorem

7.11 Geometry

1. How does the geometry of neural networks look like?
2. Hanin, Rolnick - Complexity of Linear Regions in Deep Networks - 1901.09021
   They count the number of regions, or area of boundary. Also flatness of functions? Use co-area formula from geometric measure theory.

8 How to learn practical DL

- There are now many relatively easy ways to learn DL
  - Online courses. fast.ai, deeplearning.ai
  - Books: Deep learning with Python/R, by Chollet
- To practice, three main options: Cloud, cluster or local workstation
  - Cloud: end vendors like Google Cloud, Amazon AWS, Paperspace, or Crestle, Cirrascale. cca $0.5/ hour
  - Free cloud platform: Google colaboratory
  - Cluster: Penn HPC
  - Local: need Nvidia GPU, Titan XP or 1080 Ti, cca $1,000-2,000

- Software:
  - end-user: Keras
  - More developer-oriented: Tensorflow, PyTorch
- Good references (if unconventional)
  https://lilianweng.github.io/lil-log/