Special Topics in Advanced Machine Learning

Lecture 3

Anna Choromanska
achoroma@gmail.com
http://cims.nyu.edu/~achoroma/

Department of Electrical and Computer Engineering
New York University Tandon School of Engineering

Lecture outline

- Deep learning
- Feed-forward neural networks
- Back-propagation algorithm
- Convolutional neural networks (CNNs)
- Regularization
- Adversarial networks
- Autoencoders
- LISTA
- Non-convexity and optimization landscape in deep learning

Bibliography

- T. Jebara. Course notes, Machine Learning, Topic 4. (Back-propagation algorithm)
- `http://deeplearning.net/tutorial/lenet.html` (CNNs)
- N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, pages 19291958, 2014. (Regularization)
- G. Huang, Y. Sun, Z. Liu, D. Sedra, K. Q. Weinberger, Deep Networks with Stochastic Depth, CoRR, abs/1603.09382, 2016
- I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In NIPS, 2014. (Adversarial networks)
- `http://deeplearning.stanford.edu/tutorial` (Autoencoders)
- Gregor, K. and LeCun, Y., Learning fast approximations of sparse coding, ICML 2010. (LISTA)
- P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, Entropy-SGD: Biasing Gradient Descent Into Wide Valleys, CoRR, abs/1611.01838, 2016 (Non-convexity and optimization landscape in deep learning)

Provided slides are occasionally using fragments of texts/only slightly modified versions of original slides from bibliographic material. They are used for educational purposes only.

Deep learning

- **Deep Learning** = modeling high-level abstractions in data by using networks with multiple processing layers composed of multiple non-linear transformations
- **Deep Learning** = Learning Hierarchical Representations with big (even hundreds of millions of parameters) networks
- Deep = more than one stage of non-linear feature transformation
- *Examples*: feedforward nets, convolutional nets, adversarial nets, recurrent nets, Boltzmann machines, autoencoders, and more

- Typical statistical setting: number of samples >> number of parameters (one rule of the thumb states that the number of data points should be no less than some multiple, 5 or 10, of the number of adaptive parameters in the model), but deep networks typically work in slight over-parametrization, e.g. Alexnet had 60 million parameters and was trained using 1.2 million examples.

Deep learning

- State-of-the-art results: image recognition [KSH12, CMGS10], speech recognition [HDYDMJSVNSK12, GMH13], natural language processing [WCA14], video recognition [KTSLSF-F14, SZ14], etc.
- ImageNet Challenge: convolutional networks are record holders
- The best in handwriting recognition, OCR, face & people detection, object & speech recognition, semantic segmentation/scene labeling, ...
  $\Rightarrow$ **deep learning caused revolutions in these fields**
- Important in applied mathematics
- **Leading technology in Facebook, Google, Microsoft, IBM, AT&T, Baidu, Twitter, etc.** used for: search/ad ranking, content filtering, topic classification, image recognition, photo collection management, video search and indexing, copyrighted material detection, offensive content filtering, speech recognition,....
- Google supports deep learning research: they bought DeepMind for 400M GBP and spend $\sim$ **50-100M USD/year** to support it.
- Hardware companies producing "deep learning chips"
- **NVIDIA**: deep learning for autonomous driving cars (recent!!!)

Deep learning

"Deep learning impacts a wide range of signal and information processing work. Various workshops emerged in ICML, NIPS etc. as well as special issues, e.g. deep learning for speech and language processing in IEEE Transactions on Audio, Speech, and Language Processing. They have been devoted exclusively to deep learning and its applications to classical signal processing areas."

Dong Yu and Li Deng

Feed-forward neural networks

Feed-forward neural network - connections between the units do not form a cycle; information always moves one direction and it never goes backwards

Single-layer perceptron - the simplest feed-forward neural network

- consists of a single layer of output nodes and the inputs are fed directly to the outputs via a series of weights

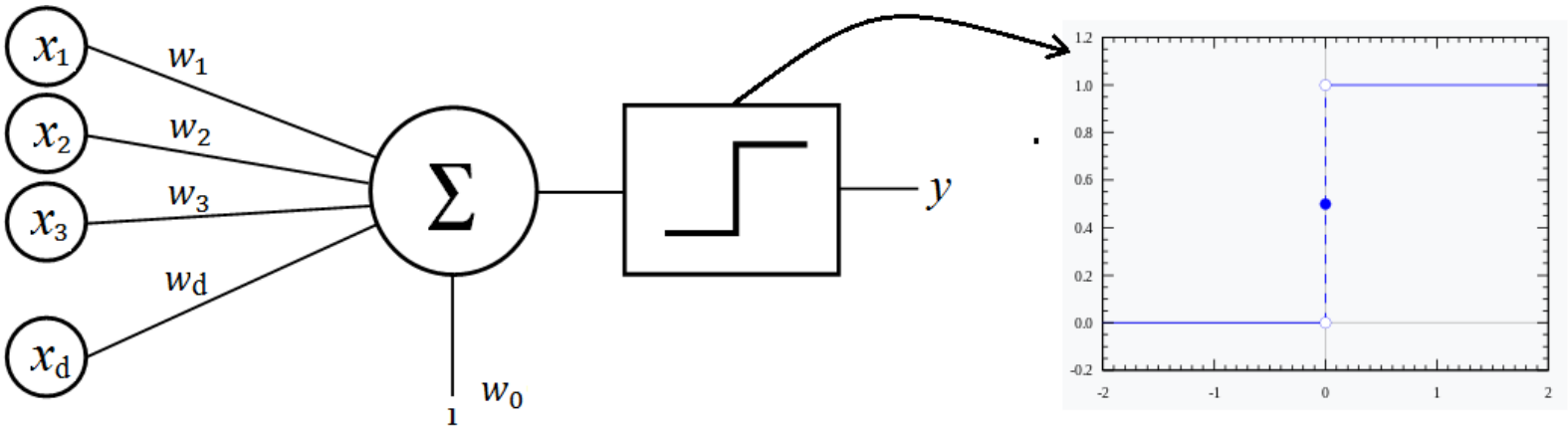


Figure: Single-layer perceptron.

- it uses unit step function as the activation function
- it is only capable of learning linearly separable patterns
- it can't handle XOR
- we discussed the learning algorithm for perceptron (see Lecture 2)

Feed-forward net with 2 or more layers is also called **multilayer perceptron**. It can handle XOR and almost any function. Biases will next be omitted.

Feed-forward neural networks

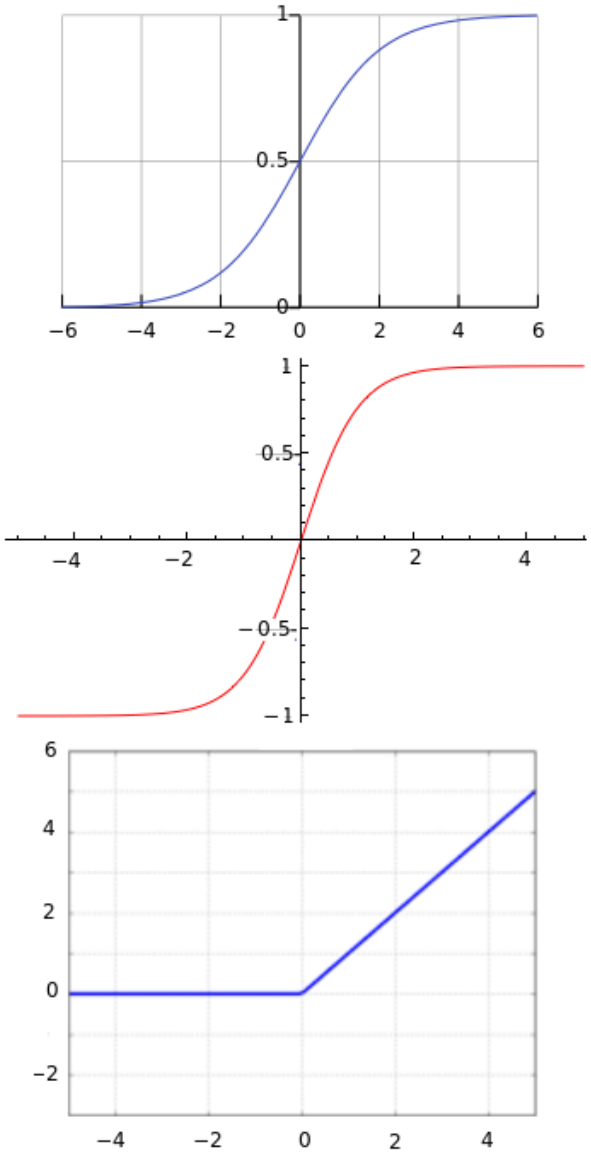


Figure: From top to bottom: sigmoid, tanh, ReLU.

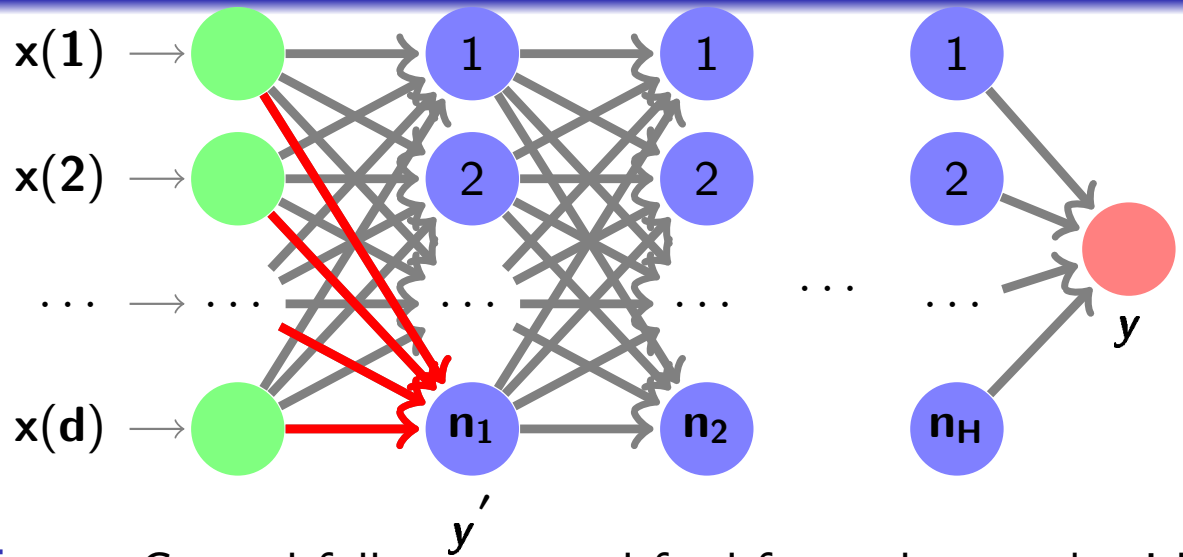


Figure: General fully-connected feed-forward network with one output (we often have more than one ouput).

Activation functions (non-linearities):

- rectifier (gives rectified linear unit, i.e. ReLU): $y' = \max(0, w^\top x)$
- sigmoid: $y' = 1/(1 + e^{-w^\top x})$
- hyperbolic tangent: $y' = \tanh(w^\top x)$
- softmax: $y' = e^{w^\top x} / \sum_{k=1}^{K} e^{w_k^\top x}$ - useful mostly in the output layer as it converts a raw value into a probability (provides certainty)
- leaky and noisy ReLU, softplus, signum, ...

Back-propagation algorithm

Training deep neural networks is an iterative scheme, where after seeing each mini-batch of examples the network adjusts its weights based on computed error terms (computed based on network output and target output). In each iteration it consists of

- forward propagation - the network computes the outputs for a given input mini-batch
- backward propagation - the "difference" between obtained outputs and target outputs is specifically back-propagated from the output to the input and the weights are updated based on computed gradients

Back-propagation algorithm

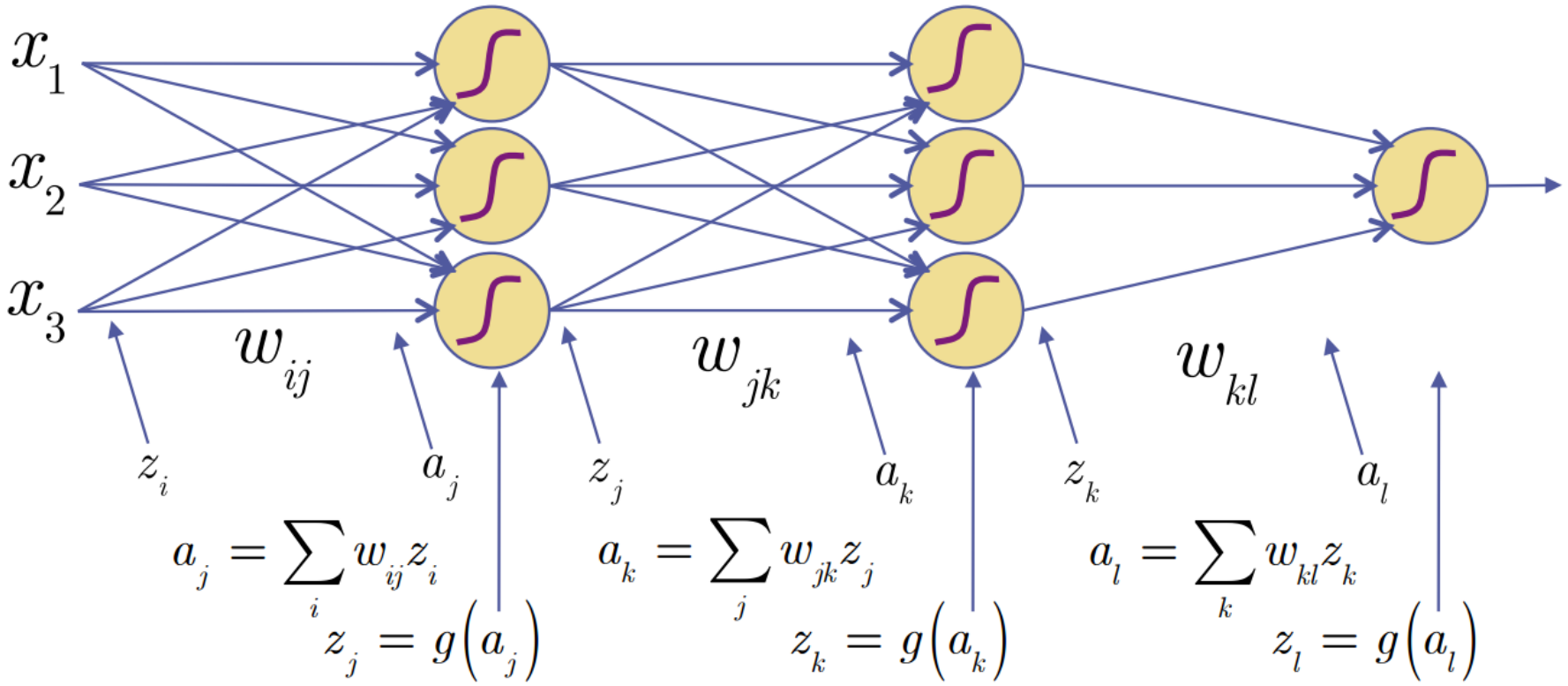


Figure: Figure from Tony Jebara's class notes.

Cost function, in case of squared loss, becomes:

$$R(\theta) = \frac{1}{N}\sum_{n=1}^{N} L(y^n - f(x^n)) = \frac{1}{N}\sum_{n=1}^{N} \underbrace{\frac{1}{2}(y^n - f(x^n))^2}_{L^n}$$

$$= \frac{1}{N}\sum_{n} \frac{1}{2}\left(y^n - g\left(\sum_k w_{k\ell} g\left(\sum_j w_{jk} g\left(\sum_i w_{ij} x_i^n\right)\right)\right)\right)^2$$

Back-propagation algorithm

- Compute first output layer derivative:

$$\begin{aligned}\frac{\partial R}{\partial w_{k\ell}} &= \frac{1}{N}\sum_n \left(\frac{\partial L^n}{\partial a_\ell^n}\right)\left(\frac{\partial a_\ell^n}{\partial w_{k\ell}}\right) \quad \text{by chain rule} \\ &= \frac{1}{N}\sum_n \left(\frac{\partial \frac{1}{2}\left(y^n - g\left(a_\ell^n\right)\right)^2}{\partial a_\ell^n}\right)\left(\frac{\partial a_\ell^n}{\partial w_{k\ell}}\right) \\ &= \frac{1}{N}\sum_n [-(y^n - z_\ell^n)g^{'}(a_\ell^n)]z_k^n = \frac{1}{N}\sum_n \delta_\ell^n z_k^n\end{aligned}$$

- Compute hidden layer derivative next:

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N}\sum_n \left(\frac{\partial L^n}{\partial a_k^n}\right)\left(\frac{\partial a_k^n}{\partial w_{jk}}\right) \quad \text{by chain rule}$$

$$= \frac{1}{N}\sum_n \left(\sum_\ell \frac{\partial L^n}{\partial a_\ell^n}\frac{\partial a_\ell^n}{\partial a_k^n}\right)\left(\frac{\partial a_k^n}{\partial w_{jk}}\right) \quad \text{by multivariate chain rule}$$

$$= \frac{1}{N}\sum_n \left(\sum_\ell \delta_l^n \frac{\partial a_\ell^n}{\partial a_k^n}\right) z_j^n = \frac{1}{N}\sum_n \left[\sum_\ell \delta_\ell^n w_{k\ell} g^{'}(a_k^n)\right] z_j^n = \frac{1}{N}\sum_n \delta_k^n z_j^n$$

Back-propagation algorithm

- For yet another previous layer do exactly the same:

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N}\sum_n \left(\frac{\partial L^n}{\partial a_j^n}\right)\left(\frac{\partial a_j^n}{\partial w_{ij}}\right) \quad \text{by chain rule}$$

$$= \frac{1}{N}\sum_n \left(\sum_k \frac{\partial L^n}{\partial a_k^n}\frac{\partial a_k^n}{\partial a_j^n}\right)\left(\frac{\partial a_j^n}{\partial w_{ij}}\right) \quad \text{by multivariate chain rule}$$

$$= \frac{1}{N}\sum_n \left[\sum_k \delta_k^n w_{jk} g'(a_j^n)\right] z_i^n = \frac{1}{N}\sum_n \delta_j^n z_i^n$$

- Last zs are the inputs.
- Update weights according to gradient descent with a small step η:

$$w_{ij}^{t+1} = w_{ij}^t - \eta\frac{\partial R}{\partial w_{ij}} \quad w_{jk}^{t+1} = w_{jk}^t - \eta\frac{\partial R}{\partial w_{jk}} \quad w_{k\ell}^{t+1} = w_{k\ell}^t - \eta\frac{\partial R}{\partial w_{k\ell}}$$

- Difficulties with neural networks, e.g. hard to interpret, what are hidden layers doing?, ...
- Digits LeNet: `http://yann.lecun.com/exdb/lenet/index.html`

Back-propagation algorithm

Backpropagation through ReLU is very simple:

- ReLU is like a switch during backpropagation.
- Let the action fo ReLU be denoted as $q = \max(0, x)$. Then $\frac{\partial q}{\partial x} = 1$ if $x > 0$. By chain rule, $\frac{\partial q}{\partial w} = \frac{\partial q}{\partial x} \cdot \frac{\partial x}{\partial w}$.
- ReLU unit lets the gradient pass unchanged if its input was greater than 0, and kills if its input was less than zero during the forward pass.

Convolutional neural networks (CNNs)

Convolutional Neural Networks (CNN) - biologically-inspired variants of MLPs.

Visual cortex:

- contains a complex arrangement of cells sensitive to small sub-regions of the visual field, called a receptive field
- sub-regions are tiled to cover the entire visual field
- cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images
- two basic cell types are:
  - simple cells - they respond maximally to specific edge-like patterns within their receptive field
  - complex cells - they have larger receptive fields and are locally invariant to the exact position of the pattern

CNNs in some sense emulate the animal visual cortex, a powerful natural visual processing system

Convolutional neural networks (CNNs)

Sparse connectivity in CNNs:

- local connectivity pattern between neurons of adjacent layers, i.e. the inputs of hidden units in layer m are from a subset of units in layer $m-1$, units that have spatially contiguous receptive fields

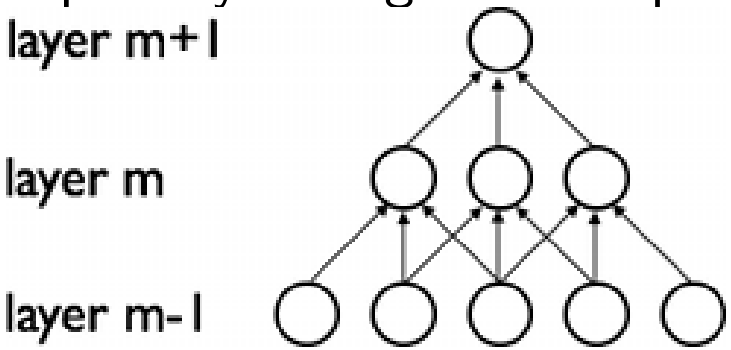


Figure: Figure from `http://deeplearning.net/tutorial/lenet.html`. Sparse connectivity in CNNs. Let layer $m-1$ be the input retina. Units in layer m have receptive fields of width 3 in the input retina (they are connected to 3 adjacent neurons in the retina layer); units in layer $m+1$ have receptive field of width 3 w.r.t. the layer below, but of width 5 w.r.t. the input retina.

- each unit is unresponsive to variations outside of its receptive field with respect to the retina - spatially-local correlation is exploited this way
- the architecture ensures that the learned filters produce the strongest response to a spatially local input pattern
- stacking many such layers leads to (non-linear) filters that become increasingly global, i.e. responsive to a larger region of pixel space

Convolutional neural networks (CNNs)

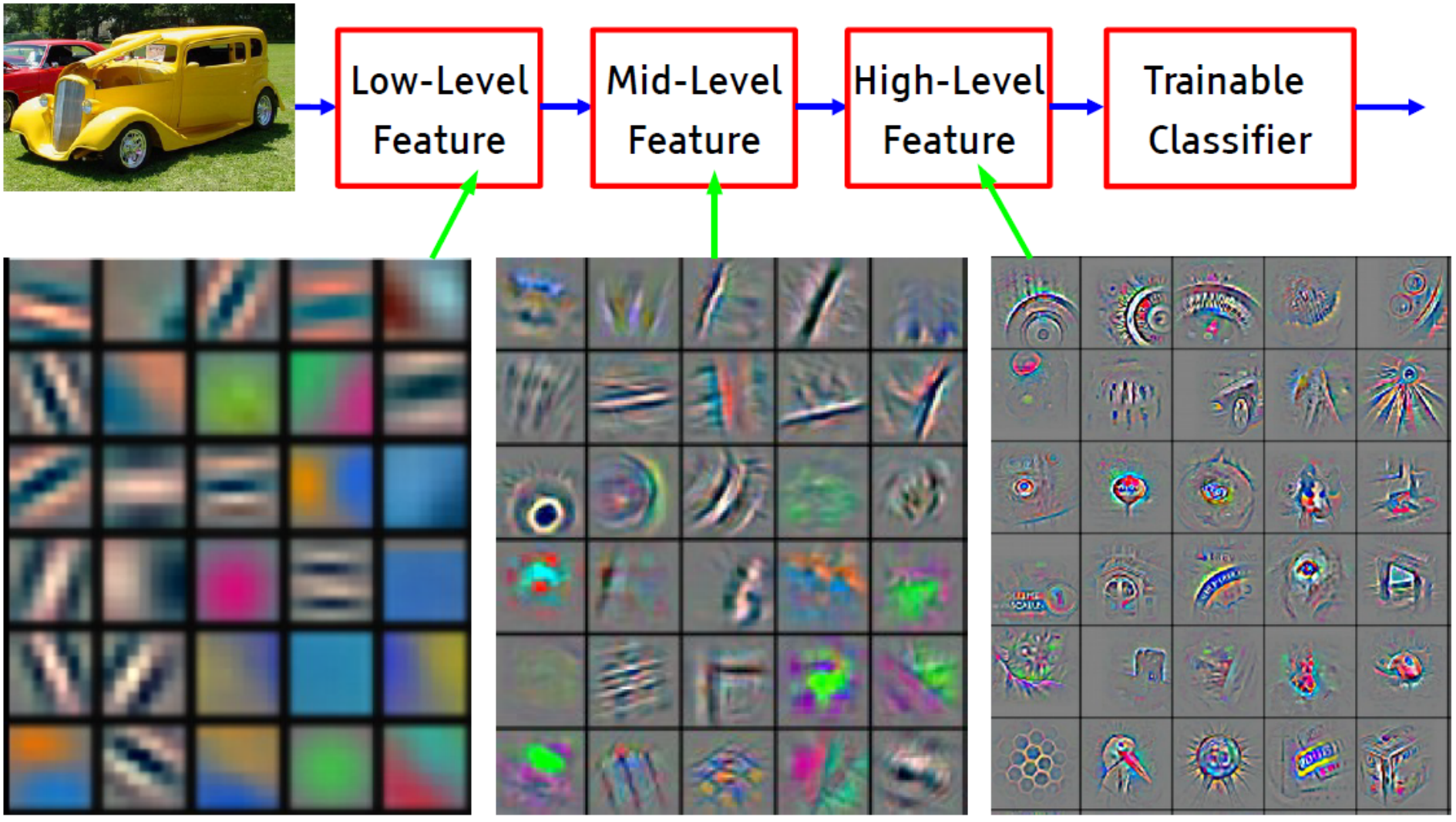


Figure: Figure from Y. LeCun Presentation, "What's wrong with deep learning?". Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013].

Convolutional neural networks (CNNs)

Shared weights in CNNs:

- each filter is replicated across the entire visual field
- replicated units share the same parameterization (weight vector and bias) and form a **feature map**

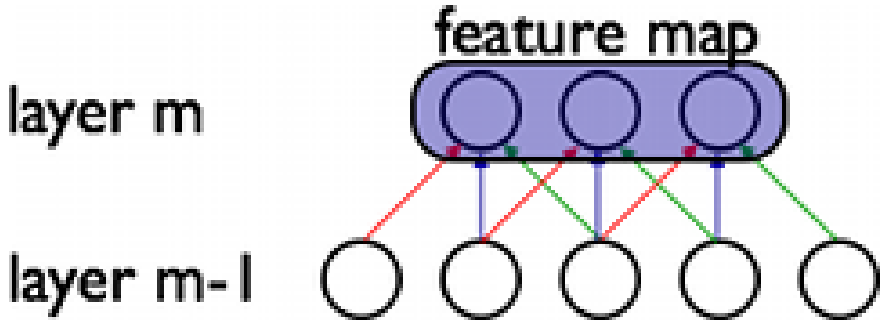


Figure: Figure from `http://deeplearning.net/tutorial/lenet.html`. Shared weights in CNNs. 3 hidden units belong to the same feature map. Weights of the same color are shared - constrained to be identical.

- gradient descent is slightly modified: the gradient of a shared weight is simply the sum of the gradients of the parameters being shared
- replicating units allows for features to be detected regardless of their position in the visual field
- weight sharing increases learning efficiency by greatly reducing the number of free parameters being learned and leads to better generalization on vision problems

Convolutional neural networks (CNNs)

Recall convolution:

- For 1D signal:
  $f(n) * g(n) = \sum_{u=-\infty}^{\infty} f(u)g(n-u) = \sum_{u=-\infty}^{\infty} f(n-u)g(u)$
- For 2D signal:
  $f(m,n) * g(m,n) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(u,v)g(m-u,n-v)$

Feature map - obtained by repeated application of a function across sub-regions of the entire image, in other words, by convolution of the input image with a linear filter, adding a bias term and then applying a non-linear function.

- h^k - k-th feature map at a given layer
- W^k, b_k - weights and bias of the filters of this k-th feature map
- q - non-linearity used by the network

Then

$$h_{ij}^k = q((W^k * x)_{ij} + b_k).$$

Convolutional neural networks (CNNs)

- in practice, each hidden layer is composed of multiple feature maps, $\{h^{(k)}, k = 0 \dots K\}$ to obtain richer data representation
- the weights W of a hidden layer can be represented in a 4D tensor containing elements for every combination of:
  - destination feature map
  - source feature map
  - source vertical position
  - source horizontal position
- for example, $W_{ij}^{k\ell}$ denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i, j) of the ℓ-th feature map of layer $(m - 1)$
- the biases b can be represented as a vector containing one element for every destination feature map

Convolutional neural networks (CNNs)

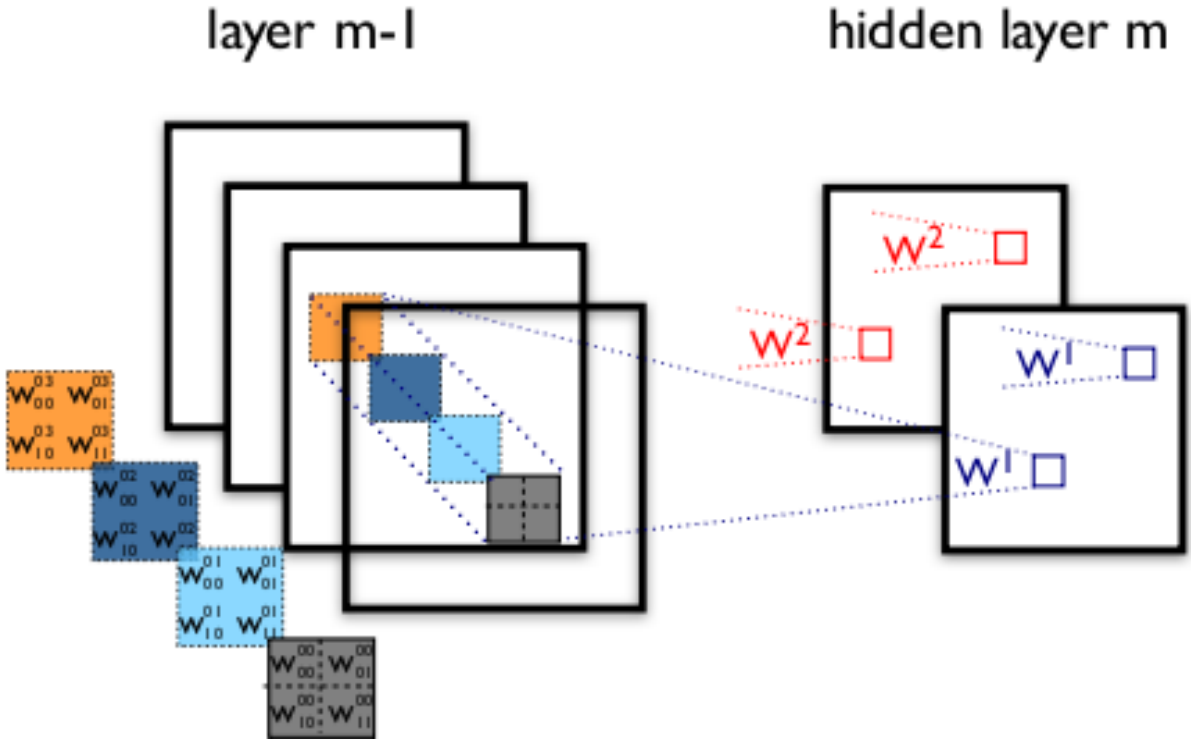


Figure: Figure from `http://deeplearning.net/tutorial/lenet.html`. Two exemplary layers of a CNN. Layer $m-1$ contains four feature maps and hidden layer m contains two feature maps (h^0 and h^1). Pixels (neuron outputs) in h^0 and h^1 (blue and red squares) are computed from pixels of layer $(m-1)$ which fall within their 2×2 receptive field in the layer below (shown as colored rectangles). Receptive field spans all four input feature maps. The weights W^0 and W^1 of h^0 and h^1 are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Convolutional neural networks (CNNs)

Max-pooling in CNNs:

- a form of non-linear down-sampling
- it partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value
- it reduces computation for upper layers
- it provides a form of translation invariance. *Example*: Cascade a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2×2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3×3 window, this jumps to $5/8$.
- since it provides additional robustness to position, it is a smart way of reducing the dimensionality of intermediate representations

Convolutional neural networks (CNNs)

LeNet family of models:

- lower-layers: alternating convolution and max-pooling layers (lower-layers operate on 4D tensors (mini-batch size $\times$ color $\times$ height $\times$ width) and are then flattened to a 2D matrix (mini-batch size $\times$ color $\cdot$ height $\cdot$ width) of rasterized feature maps to be compatible with subsequent MLP)
- upper-layers: fully-connected and correspond to a traditional MLP (hidden layer + logistic regression)
- the input to the first fully-connected layer is the set of all features maps at the layer below

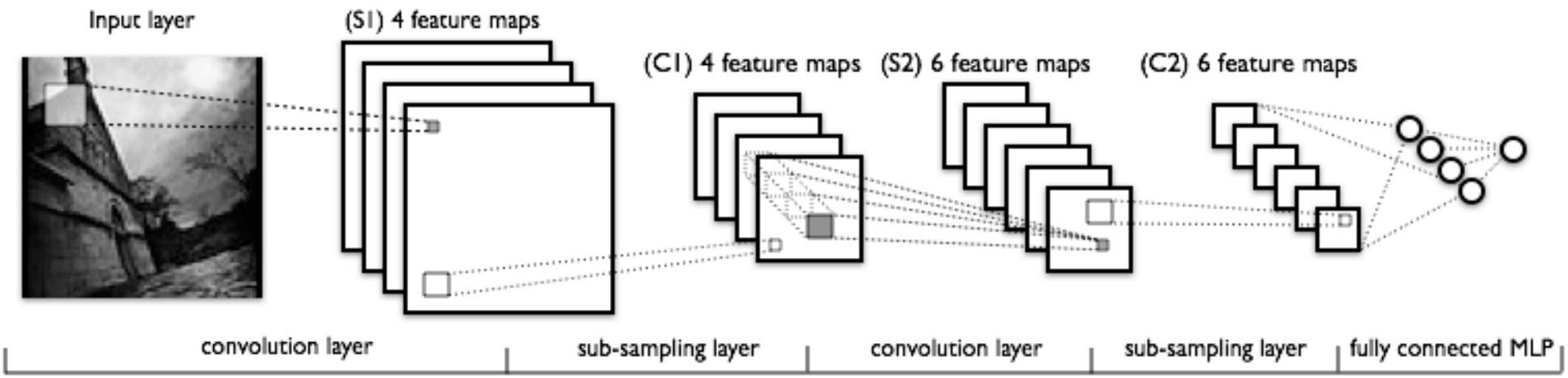


Figure: Figure from `http://deeplearning.net/tutorial/lenet.html`. LeNet model.

Convolutional neural networks (CNNs)

Choosing hyper-parameters:

- CNNs have more hyper-parameters than standard MLPs
- Number of filters (general):
  - computing the activations of a single convolutional filter is much more expensive than with traditional MLPs
  - assume layer $(\ell - 1)$ contains $K^{\ell-1}$ feature maps and $M \times N$ pixel positions, and there are K^{ℓ} filters at layer ℓ of shape $m \times n$ (we assume stride 1×1)
    - computing a feature map (applying an $m \times n$ filter at all $(M - m + 1) \times (N - n + 1)$ pixel positions where the filter can be applied) costs $(M - m + 1) \times (N - n + 1) \times m \times n \times K^{\ell-1}$
    - the total cost is K^{ℓ} times that
    - things may be more complicated if not all features at one level are connected to all features at the previous one
  - for a standard MLP, the cost would only be $K^{\ell} \times K^{\ell-1}$ where there are K^{ℓ} different neurons at level ℓ
  - thus, the number of filters used in CNNs is typically much smaller than the number of hidden units in MLPs and depends on the size of the feature maps (itself a function of input image size and filter shapes)

Convolutional neural networks (CNNs)

- Number of filters (rules):
  - feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more:
    - to equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers
    - to preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (we could hope to get away with less when we are doing supervised learning)
    - the number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task
- Filter Shape:
  - best results on MNIST-sized images (28×28) are usually in the 5×5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12×12 or 15×15
- Max Pooling Shape:
  - typical values are 2×2 or no max-pooling; very large input images may warrant 4×4 pooling in the lower-layers (this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information)

Convolutional neural networks (CNNs)

Practical example 1: Digits recognition Architecture:

- Input image (MNIST): $1 \times 32 \times 32$
- Convolutional layer: $\underbrace{1 \to 4}_{\text{number of input and output chanels}}, \underbrace{5 \times 5}_{\text{filter size}}, \underbrace{1 \times 1}_{\text{stride}}$
- ReLU
- MaxPooling: $\underbrace{2 \times 2}_{\text{region size}}, \underbrace{2 \times 2}_{\text{stride}}$
- Convolutional layer: $4 \to 6, 5 \times 5, 1 \times 1$
- ReLU
- MaxPooling: $2 \times 2, 2 \times 2$
- Flattening (3D to 1D): $6 \times 5 \times 5 \to 150$ (we skipped mini-batch size, else it would be 4D to 2D as explained before)
- Dropout (with probability 0.5)
- Fully connected layer: $150 \to 128$
- ReLU
- Fully connected layer: $128 \to \underbrace{10}_{\text{output}}$

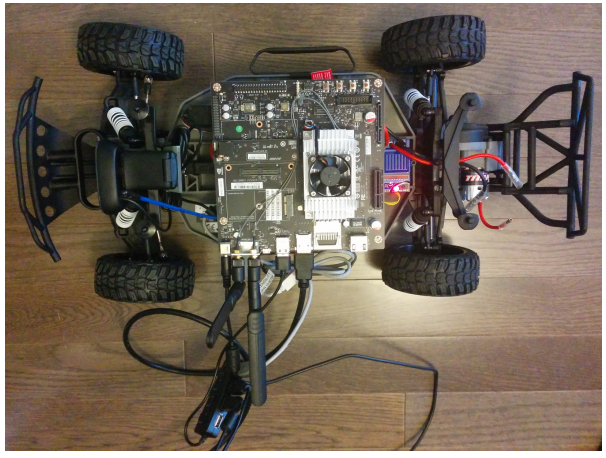
Convolutional neural networks (CNNs)

Figure: Video from `http://yann.lecun.com/exdb/lenet/index.html`. LeNet-5 model for digit recognition.

Convolutional neural networks (CNNs)

Practical example 2: Road segmentation

We have autonomous car steered with with GPU-based embedded system (NVIDIA Tegra TX1)



Convolutional neural networks (CNNs)

- Fully connected layer = convolutional layer with filter and stride 1×1.
- Fully connected layers are expensive and often redundant. There are trends in modern networks to reduce their size or eliminate them.
- Another problem with deep networks is a **vanishing gradient**. In each iteration of training with backpropagation, each of the network's weights is updated proportionally to the gradient of the error function with respect to the current weight. Many activation functions, e.g. hyperbolic tangent, have gradients in the range $(1, 1)$ or $[0, 1)$. Backpropagation computes gradients by the chain rule, thus computing the gradient of earlier layers requires multiplication of many such small numbers. As a result, the gradient (error signal) decreases exponentially and the earlier layers train very slowly.

  Also, as the gradient information is back-propagated, repeated multiplication or convolution with small weights renders the gradient information ineffectively small in earlier layers. [Huang et al., 2016]

  Various techniques help to avoid this problem, e.g. skip layers and **batch normalization** as for example in ResNets.

Regularization

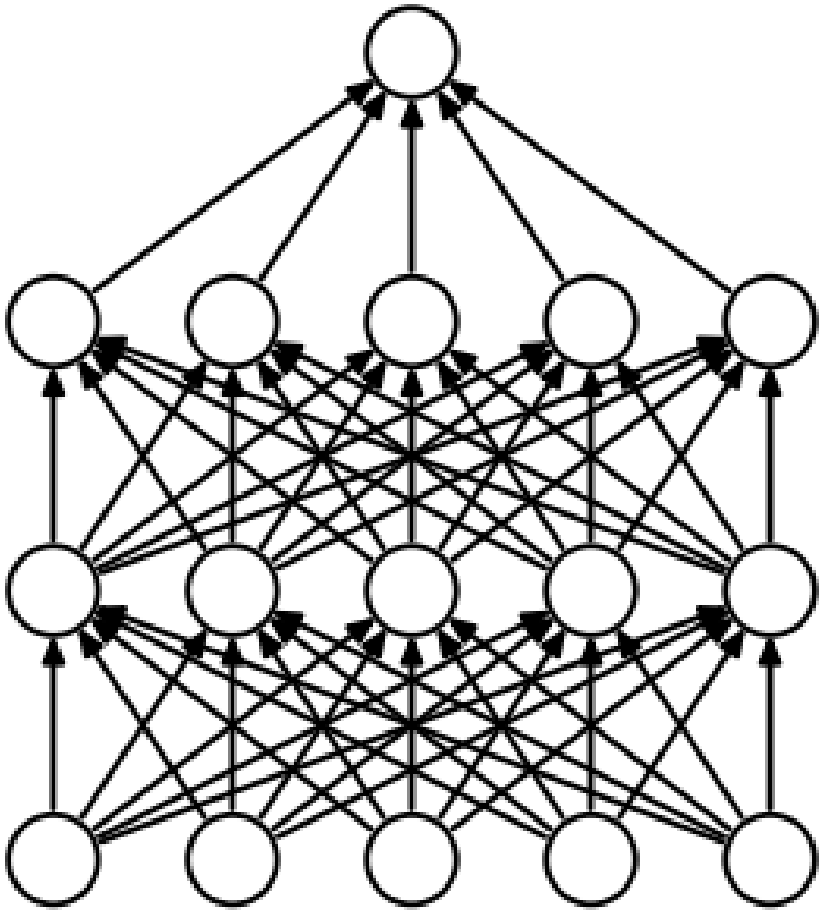
Selected regularization techniques in deep learning:

- **Pooling** - we discussed that already

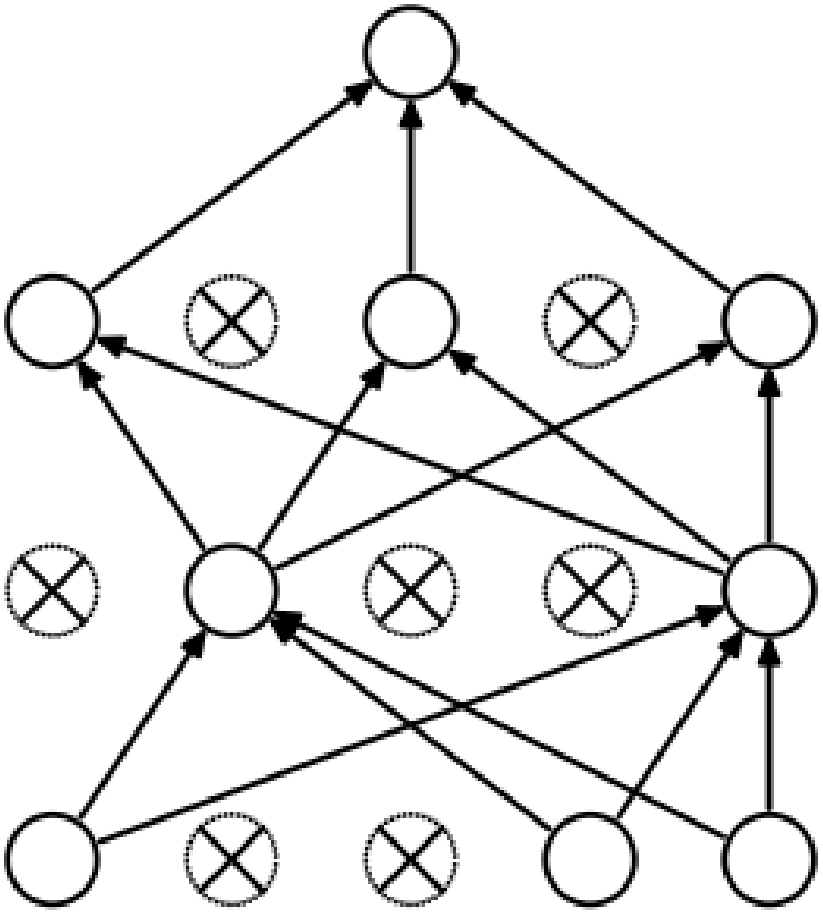
Deep neural nets have i) large number of parameters, ii) suffer from overfitting, iii) are also slow to use thus dealing with overfitting by combining the predictions of many different large neural nets at test time is prohibitive.

- **Dropout** - a technique for addressing the above problem: randomly drop units (along with their connections) from the neural network during training to prevent units from co-adapting too much (co-adapting: the learning results may overfit to the pre-defined structure of the network when the weights are simultaneously learned as they may co-adapt, i.e. a weight may converge to a certain value because it dependents on the values of some other weights). Dropout significantly reduces overfitting and gives major improvements over other regularization methods.
  - Training: dropout samples from an exponential number of different "thinned" networks.
  - Testing: approximating the effect of averaging the predictions of all these thinned networks is done by using a single unthinned network that has smaller weights.

Regularization



(a) Standard Neural Net



(b) After applying dropout.

Figure: Figure from [Srivastava et al 2014]. Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Regularization

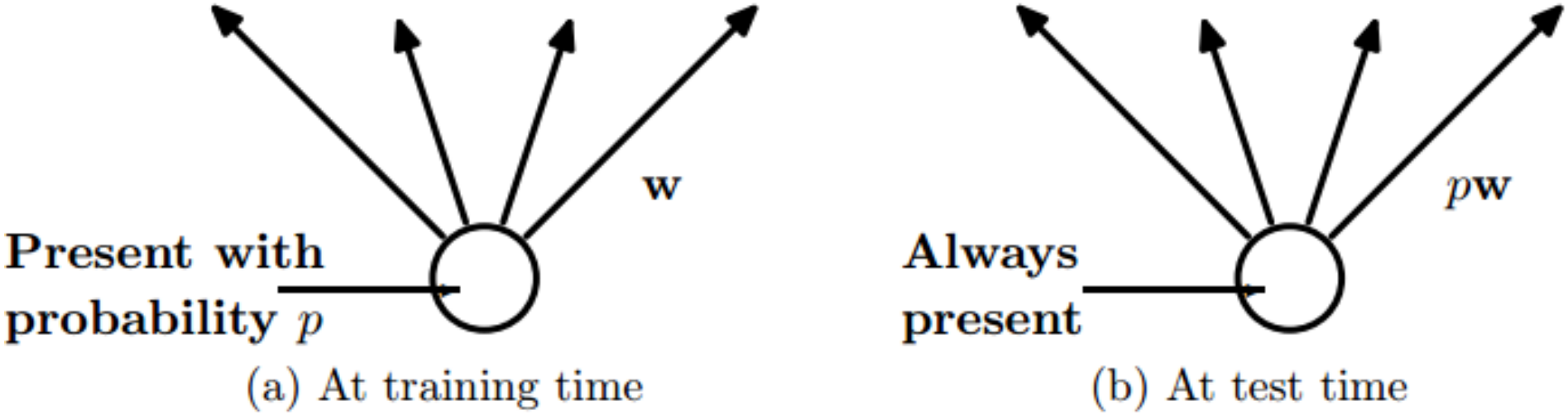


Figure: Figure from [Srivastava et al 2014]. Left: A unit at training time that is present with probability p and is connected to units in the next layer with weights w. Right: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

Adversarial networks

Generative adversarial networks (GANs) - a new framework for estimating generative models (it is a model for randomly generating observable data values) via an adversarial process.

GAN framework involves training two models:

- generative model G that captures the data distribution
- discriminative model D that estimates the probability that a sample came from the training data rather than G

Training of both models correspond to a minimax two-player game:

- G is trained to maximize the probability of D making a mistake (fool D)
- D is trained to maximize the probability of assigning a correct label to both training examples and samples from G

In the space of arbitrary functions G and D, a unique solution exists:

- G recovering the training data distribution
- D equal to $1/2$ everywhere

G and D are defined by multilayer perceptrons and can be trained with backpropagation (no need for Markov chains or unrolled approximate inference nets for training or generation of samples). Sampling from G is done using forward propagation.

Adversarial networks

Notation:

- θ_g, θ_d - parametrization of networks G and D respectively
- p_g - generator's distribution (it is updated during training of G)
- $p_z(z)$ - prior distribution on input noise variables z (usually uniform)
- $G(z, \theta_g)$ - mapping to data space done by G (G converts p_z to p_g; while training G we expect p_g to converge to p_{data})
- $D(x, \theta_d)$ - scalar - probability that x came from the data rather than p_g

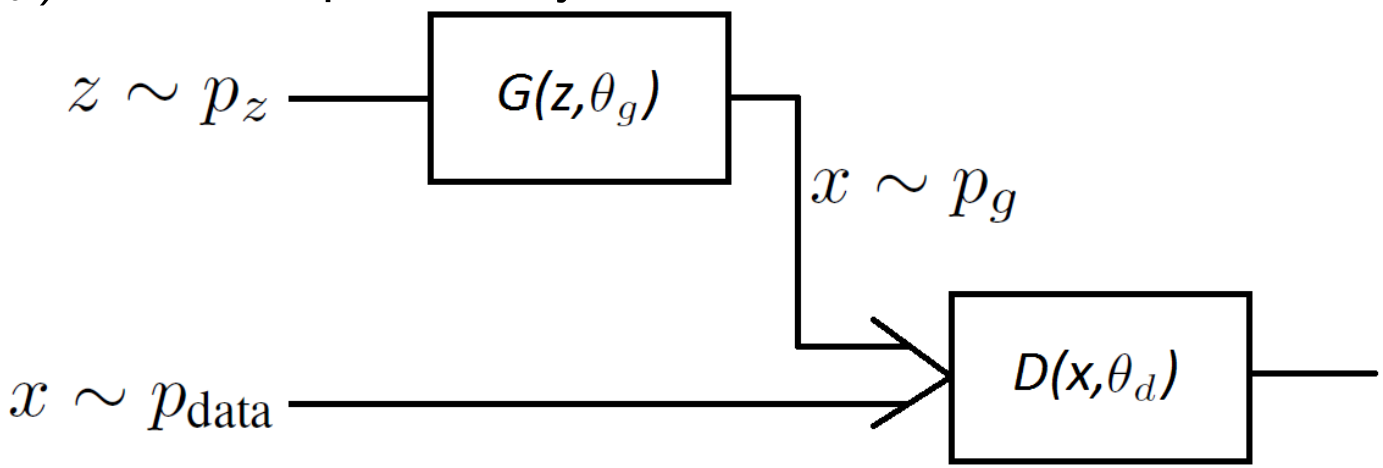
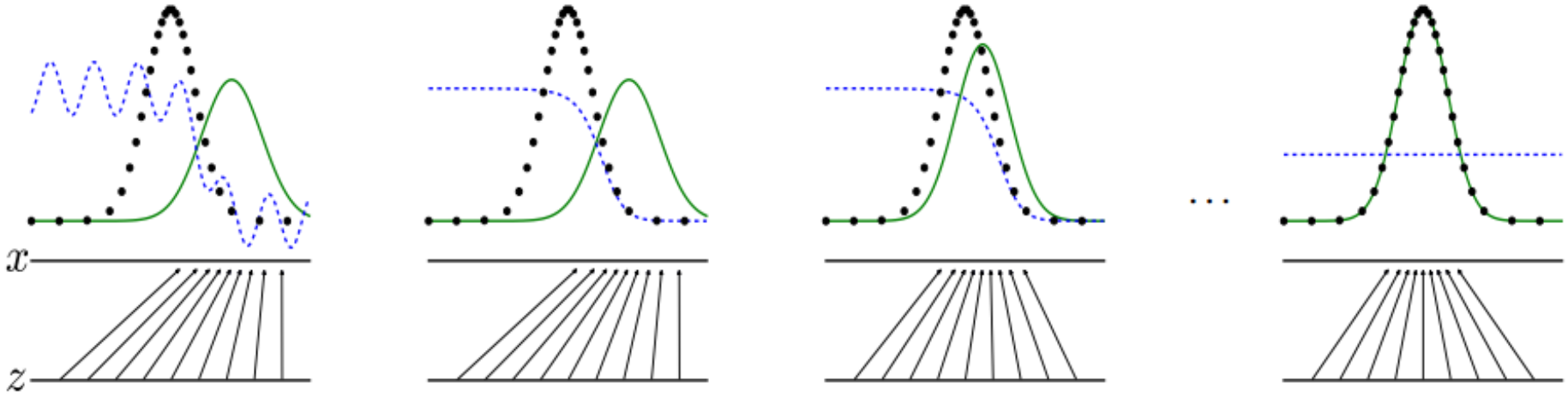


Figure: Illustration of the adversarial scheme.

Adversarial nets represent a limited family of p_g distributions via the function $G(z, \theta_g)$ and we optimize θ_g rather than p_g itself. Thus, there is no closed-form formula for p_g that would be assumed. G 'realizes' the process of sampling from some p_g and this is learned, not the form of p_g.

Adversarial networks



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

Figure: Figure from Goodfellow et al., NIPS 2014. Illustration of training GANs.

Adversarial networks

Training GANs:

- G is trained to minimize $\log(1 - D(G(z, \theta_g), \theta_d))$ and D is trained to maximize the probability of assigning a correct label to both training examples and samples from G. Using the language of minimax games, D and G play a two-player minimax game with value function $V(\theta_g, \theta_d)$:

$$\min_{\theta_g} \max_{\theta_d} V(\theta_g, \theta_d)$$
$$= \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(D(x, \theta_d))] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z, \theta_g), \theta_d))]$$

- Optimizing D to completion in the inner loop of training is computationally prohibitive, and on finite datasets overfits. Instead, alternating between k steps of optimizing D and one step of optimizing G results in D being maintained near its optimal solution, so long as G changes slowly enough.
- Since early in learning, when G is poor, D can reject samples with high confidence as they are clearly different from the training data, $\log(1 - D(G(z, \theta_g), \theta_d))$ saturates. Thus instead it is better to maximize $\log(D(G(z, \theta_g), \theta_d))$ to train G.

Adversarial networks

Minibatch stochastic gradient descent training of generative adversarial nets (from Goodfellow et al., NIPS 2014):

Set $k = 1$ (k - number of steps to train D at each iteration).

for number of training iteration **do**

  for k steps **do**

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

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

  end for

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

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

end for

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

Adversarial networks

Theorem 1

For G fixed, the optimal discriminator D is

$$D^*_G(x, \theta_d) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}.$$

Theorem 2

The minimax game has a global optimum for $p_g = p_{data}$, i.e. the generative model perfectly replicating the data generating process.

Theorem 3 (Convergence of the algorithm)

If G and D have enough capacity, and at each step of the adversarial optimization algorithm, D is allowed to reach its optimum given G, and p_g is updated to improve the criterion

$$\mathbb{E}_{x \sim p_{data}}[\log(D^*_G(x, \theta_d))] + \mathbb{E}_{x \sim p_g}[\log(1 - D^*_G(x, \theta_d))],$$

then p_g converges to p_{data}.

Adversarial networks

Figure: Figure from Goodfellow et al., NIPS 2014. Visualization of samples from the model distributions. Rightmost column shows the nearest training example of the neighboring sample, in order to demonstrate that the model has not memorized the training set. Samples are uncorrelated. a) MNIST b) TFD c) CIFAR-10 d) CIFAR-10.

Autoencoders

Autoencoders - used in unsupervised learning setting, i.e. when we have access to unlabeled training examples $\{x^{(1)}, x^{(2)}, \dots\}$ $(x^{(i)} \in \mathbb{R}^n)$. An autoencoder neural network is an unsupervised learning algorithm that sets the target values to be equal to the inputs, i.e. $y^{(i)} = x^{(i)}$.

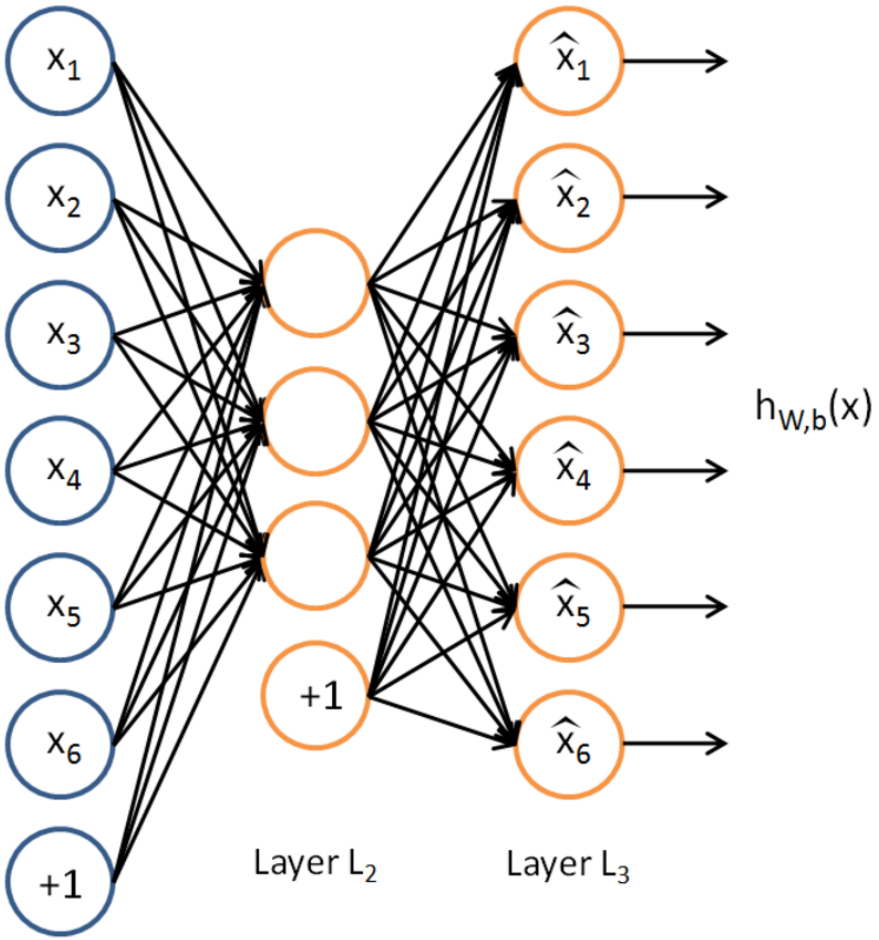


Figure: Figure from `http://deeplearning.stanford.edu/tutorial`. The scheme of an exemplary autoencoder which tries to learn a function $h_{W,b}(x) \approx x$.

Autoencoders

- The identity function is a trivial function to learn, but placing constraints on the network, such as by limiting the number of hidden units, we can discover interesting structure about the data.

  Example: suppose the inputs x are the pixel intensity values from a 10×10 (100 pixels) so $n = 100$, and there are $s_2 = 50$ hidden units in layer L_2. Note that also $y \in \mathbb{R}^{100}$.
  - Limiting the number of hidden units to 50 forces the network to learn a compressed representation of the input, i.e. given only the vector of hidden unit activations of dimension 50 ($a^{(2)} \in \mathbb{R}^{50}$) it must try to reconstruct the input of dimension 100 - not difficult when there is a structure in the data like in case of images, e.g. if some of the input features are correlated, but difficult when the input is completely random.
  - This simple autoencoder often ends up learning a low-dimensional representation very similar to PCAs.
- Even when the number of hidden units is large (even greater than the number of input pixels), we can still discover interesting structure by imposing other constraints on the network like a sparsity constraint on the hidden units.

Autoencoders

Sparsity constraint:

- Assume a sigmoid activation function. Neuron is "active" if its output value is close to 1, or "inactive" if its output value is close to 0. We would like to constrain the neurons to be inactive most of the time.
- Let $\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} a_j^{(2)}(x^{(i)})$ be the average activation of hidden unit j of second layer (averaged over the training set).
- Let ρ denote the sparsity parameter which is close to 0, e.g. 0.05.
- Sparsity constraint: enforce $\hat{\rho}_j = \rho$, i.e. the average activation of each hidden neuron j is close to 0.05. To satisfy this, the hidden units activations must mostly be near 0.
- Kullback-Leibler (KL) divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$ measures how different two distributions are:
  - $\mathrm{KL}(\rho||\hat{\rho}_j) = 0$ if $\hat{\rho}_j = \rho$, and otherwise it increases monotonically as $\hat{\rho}_j$ diverges from ρ.
  - KL-divergence reaches its minimum of 0 at $\hat{\rho}_j = \rho$, and and blows up to ∞ as $\hat{\rho}_j$ approaches 0 or 1.

Autoencoders

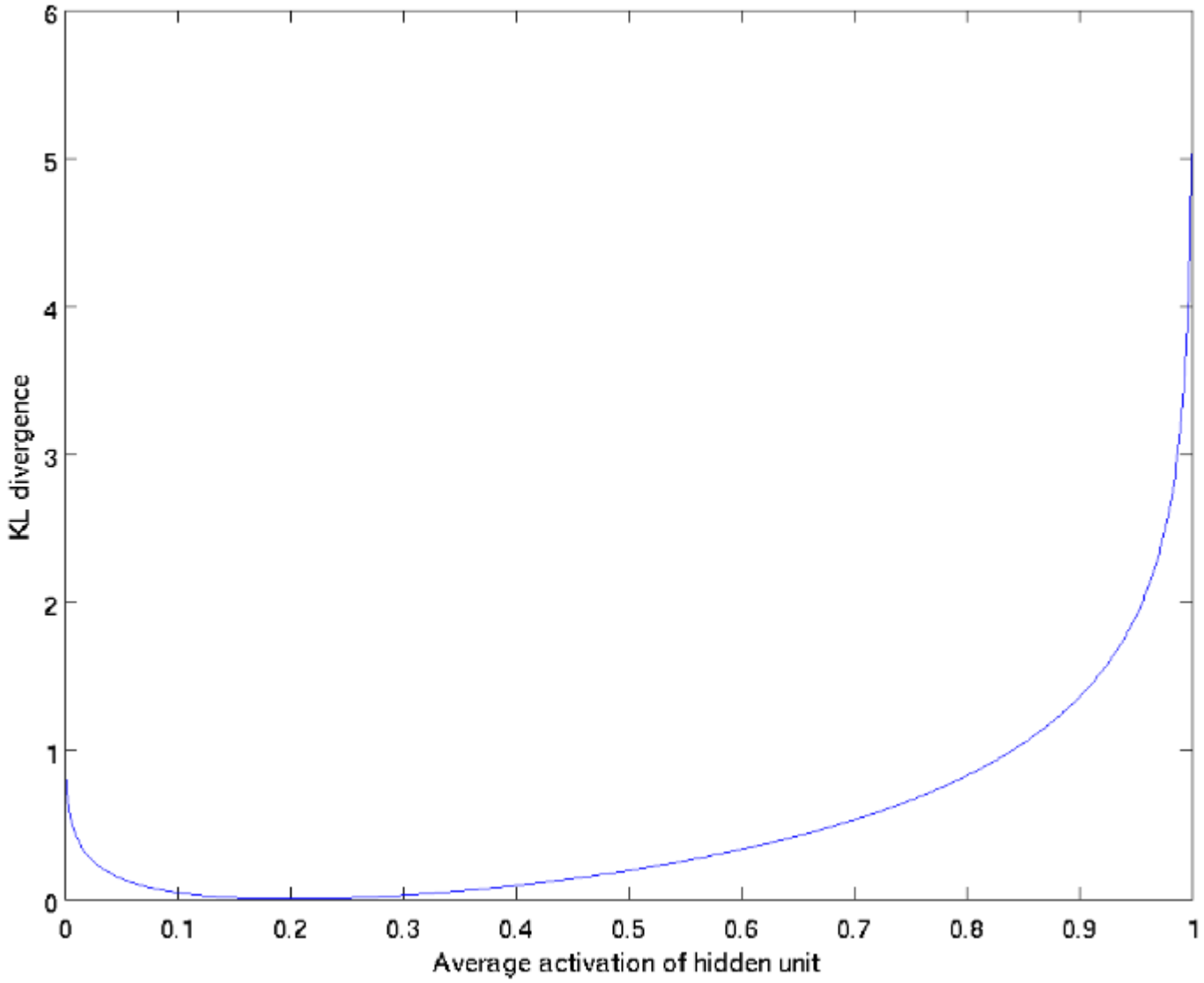


Figure: Figure from `http://deeplearning.stanford.edu/tutorial`. Plot of $\mathrm{KL}(\rho||\hat{\rho}_j)$ as a function of $\hat{\rho}_j$ when $\rho = 0.2$.

Autoencoders

- We will add an extra penalty term to our optimization objective that penalizes $\hat{\rho}_j$ deviating significantly from ρ of the form:

$$\sum_{j=1}^{s_2} \mathrm{KL}(\rho||\hat{\rho}_j) = \sum_{j=1}^{s_2} \left(\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \right),$$

  where s_2 denotes the number of neurons in the hidden layer.
- Minimizing this penalty term causes $\hat{\rho}_j$ to be close to ρ.
- Thus the objective becomes:

$$J_{\text{sparse}}(W, b) = J(W, b) + \beta \sum_{j=1}^{s_2} \mathrm{KL}(\rho||\hat{\rho}_j),$$

  where $J(W, b)$ is defined with a standard loss, like squared loss, and β controls the weight of the sparsity penalty term.
  - term $\hat{\rho}_j$ (implicitly) depends on W, b also, because it is the average activation of hidden unit j, and the activation of a hidden unit depends on the parameters W, b
- Incorporating the KL-divergence term into the derivative calculation results in a slight change in the updates of backpropagation algorithm (see `http://deeplearning.stanford.edu/tutorial`).

Autoencoders

Example: Consider 10×10 image ($n = 100$) and a net with 100 hidden units.

- Each hidden unit i computes: $a_i^{(2)} = f\left(\sum_{j=1}^{100} W_{ij}^{(1)} x_j + b_i^{(1)}\right)$, where f is the sigmoid non-linearity.
- Assume input x is norm-constrained, i.e. $\|x\|^2 = \sum_{i=1}^{100} x_i^2 \leq 1$. The input which maximally activates hidden unit i is given by setting pixel x_j (for all $j = 1, 2, \ldots, 100$) to $x_j = W_{ij}^{(1)} / \sqrt{\sum_{j=1}^{100} (W_{ij}^{(1)})^2}$.
- Each square below shows the (norm bounded) input image x that maximally actives one of 100 hidden units.

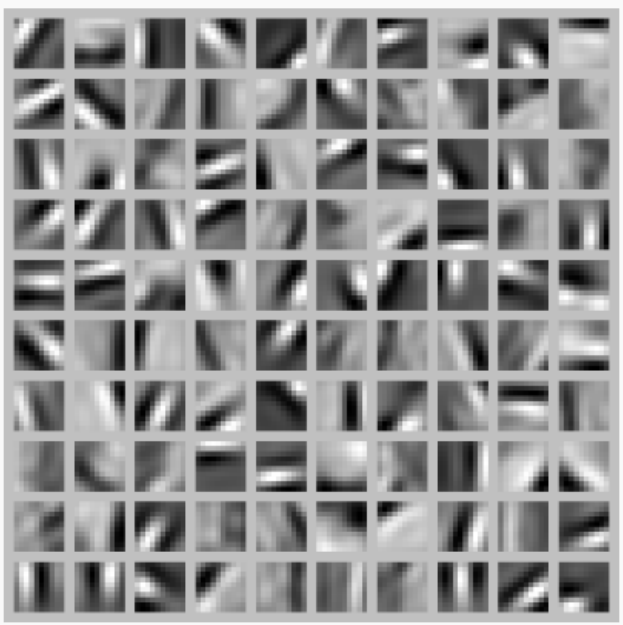


Figure: Figure from `http://deeplearning.stanford.edu/tutorial`. Different hidden units have learned to detect edges at different positions and orientations in the image. These features are useful for such tasks as object recognition and other vision tasks.

When applied to other input domains (such as audio), (sparse) autoencoder also learns useful representations/features for those domains too.

LISTA

Sparse Coding (SC)- input vectors are reconstructed using a sparse linear combination of basis vectors. SC is a method for extracting features from data.

For the inference problem, given an input vector $x \in \mathbb{R}^n$ SC aims to find the optimal sparse code vector $z^* \in \mathbb{R}^m$ that minimizes a quadratic reconstruction error with an L_1 sparsity penalty term on the code:

$$E_{W_d}(x, z) = \frac{1}{2}\|x - W_d z\|_2^2 + \alpha\|z\|_1,$$

where W_d is an $n \times m$ dictionary matrix whose columns are the (normalized) basis vectors ($m > n$ is the overcomplete case) and α is a coefficient controlling the sparsity penalty. Dictionary matrix is often learned by minimizing the average of $min_z E_{W_d}(x, z)$ over a set of training examples using SGD.

LISTA

ISTA - popular algorithm for sparse code inference.

Given an input vector x, ISTA iterates the following recursive equation to convergence:

$$z_{k+1} = h_\theta(W_e x + S z(k)) \quad z(0) = 0,$$

where $W_e = \frac{1}{L} W_d^\top$ is the filter matrix (L is a constant which must be an upper-bound on the largest eigenvalue of $W_d^\top W_d$), $S = \mathrm{I} - \frac{1}{L} W_d^\top W_d$ is the mutual inhibition matrix, and $h_\theta(V)$ is a component-wise vector shrinkage function with a vector of threshold θ, i.e. $[h_\theta(V)]_i = \mathrm{sign}(V_i)(|V_i| - \theta_i)_+$ (all thresholds are set to $\theta_i = \alpha/L$).

We will also discuss ISTA in Lecture 6.

LISTA

ISTA can be represented in a block diagram:

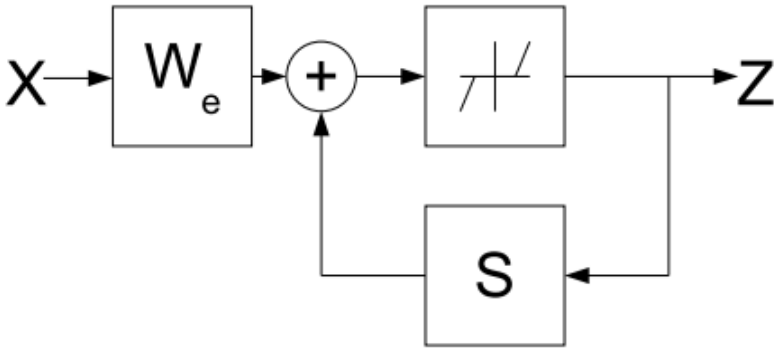


Figure: Figure from [Gregor and LeCun 2010]. Block diagram of ISTA.

Consider time-unfolded version of the ISTA block diagram, truncated to a fixed number of iterations (3 below).

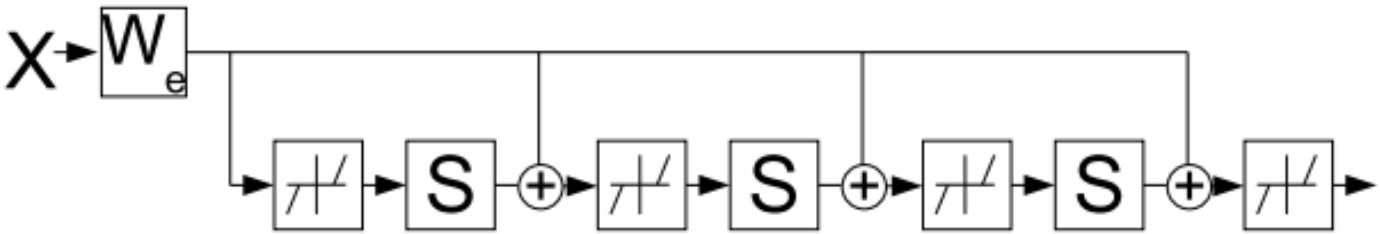


Figure: Figure from [Gregor and LeCun 2010]. Block diagram of LISTA.

Obtained algorithm is called **Learned ISTA (LISTA)**. Matrices W_e and S are learned, so as to minimize the approximation error to the optimal sparse code on a given dataset. One can impose restrictions on S to further reduce computations (e.g. keeping many terms at 0, or using a low-rank factorized form).

LISTA

LISTA gives rise to a fast encoder that can be trained to compute approximate sparse codes. More specifically, a non-linear, feed-forward predictor with a specific architecture and a fixed depth is trained to produce the best possible approximation of the sparse code.

- $z = f_e(x, W)$ - denotes the architecture of the encoder, where W denotes all trainable parameters
- $\{x^1, x^2, \ldots, x^P\}$ - training set
- $L(W) = \frac{1}{P} \sum_{p=1}^{P} \frac{1}{2} \|z^{*p} - f_e(W, x^p)\|^2$ - loss function defined as the squared error between the predicted code and the optimal code ($z^{*p} = \arg\min_z E_{W_d}(x^p, z)$ is the optimal code for sample x^p) averaged over a training set; it is minimized with SGD during training
- A simple class of encoder architectures is given by the single-layer feed-forward architecture and is of the form

$$Z = g(W_e x),$$

where W_e is an $m \times n$ trainable matrix and g is a coordinate-wise non-linearity (it can simply be the shrinkage function used in ISTA, i.e. h_θ, with trainable θ).

LISTA

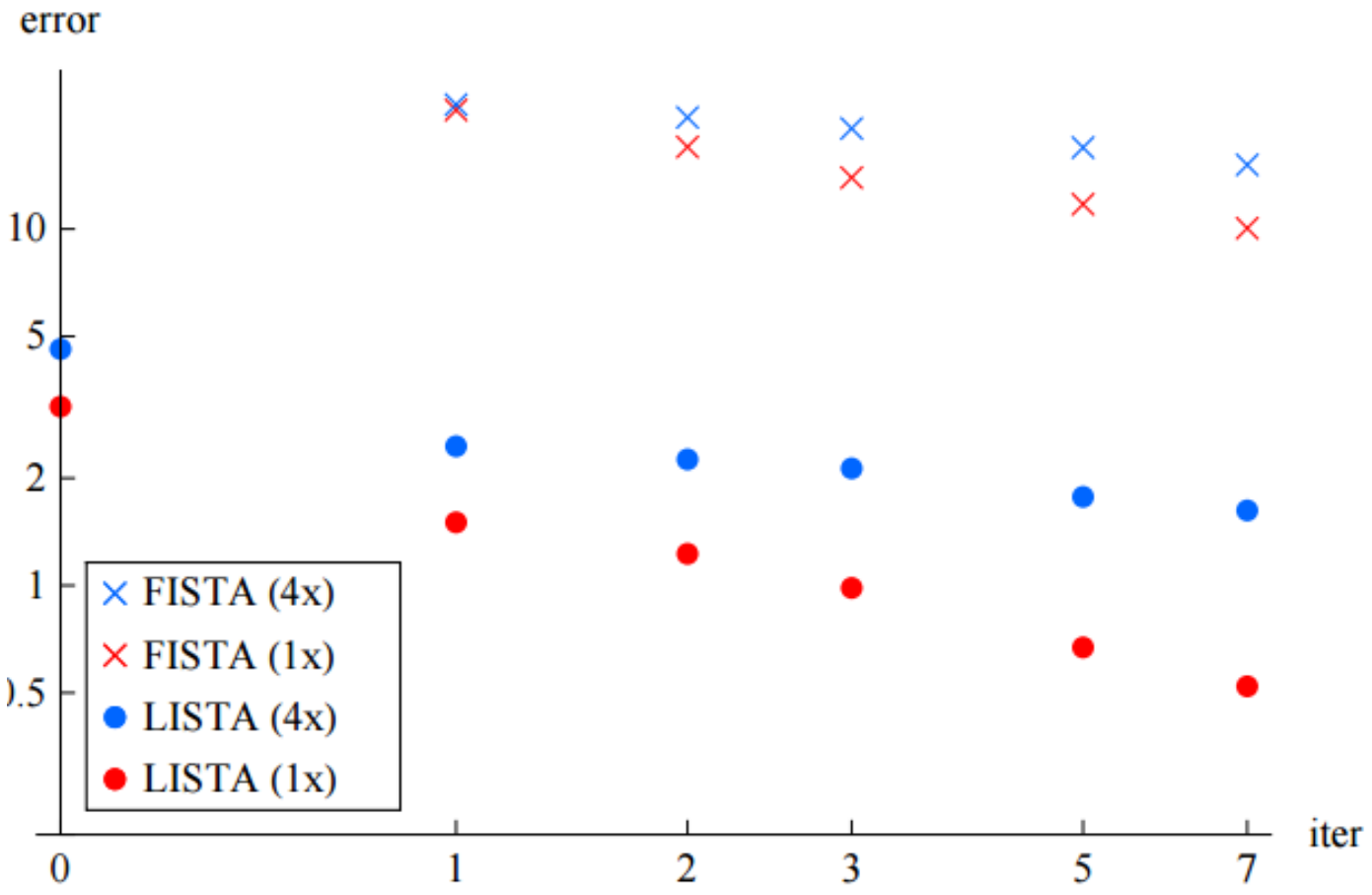
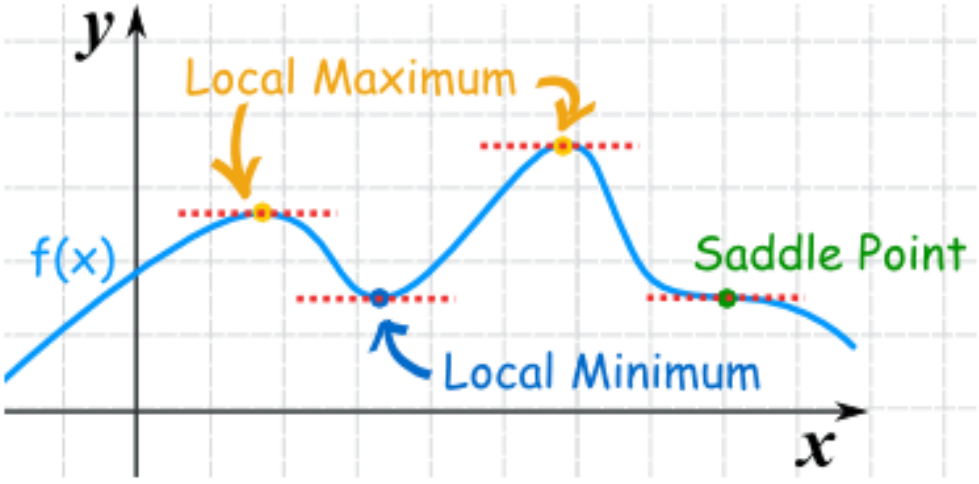


Figure: Figure from [Gregor and LeCun 2010]. Code prediction error as a function of number of iterations for FISTA, Fast version of ISTA, (crosses) and for LISTA (dots), for $m = 100$ (red) and $m = 400$ (blue). Note the logarithmic scales. iter $= 0$ corresponds to the baseline trainable encoder with the shrinkage function. It takes 18 iterations of FISTA to reach the error of LISTA with just one iteration for $m = 100$, and 35 iteration for $m = 400$.

Non-convexity and optimization landscape in deep learning

Critical point - a point where gradient is zero.



Index of critical point - # negative eigenvalues of the Hessian.

Consider function of Λ variables:

- *Local minimum* - index is 0 (Hessian is positive definite)
- *Local maximum* - index is Λ (Hessian is negative definite)
- *Saddle point* - otherwise (a point which is a maximum in some directions and a minimum in others)

Non-convexity and optimization landscape in deep learning

The existence of wide minima in the highly non-convex objective function of deep nets is suggested by the spectral analysis of the Hessian.

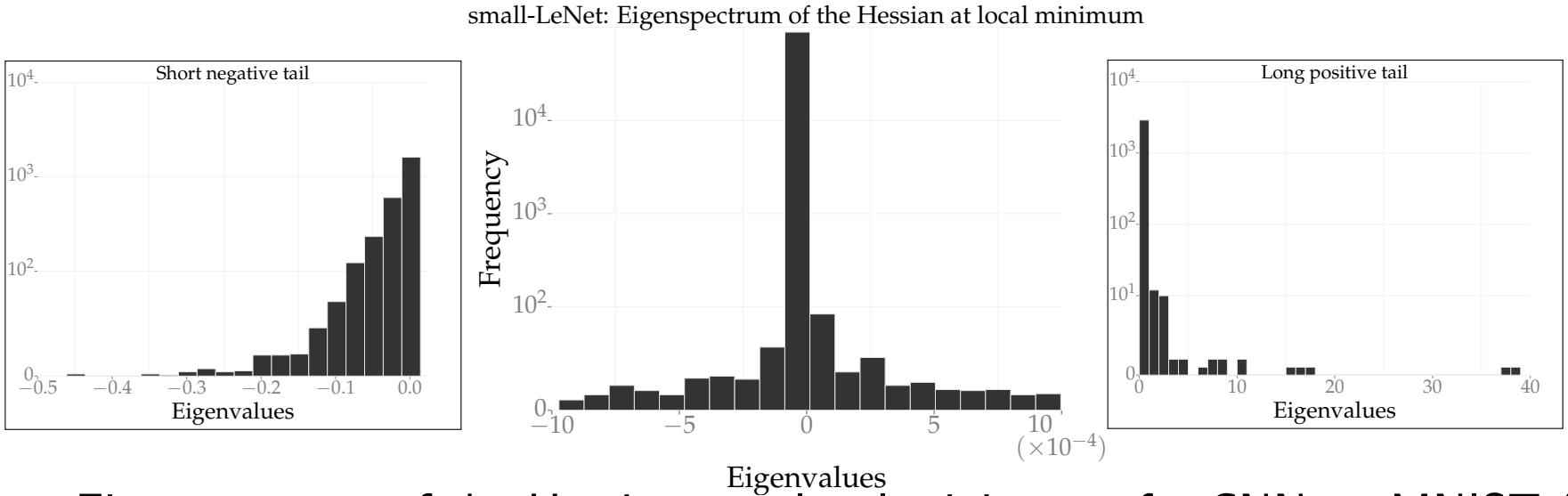


Figure: Eigenspectrum of the Hessian at a local minimum of a CNN on MNIST (two independent runs). **Remark:** Central: eigenvalues in a small neighborhood of zero; Left and right: the entire tails of the eigenspectrum.

(i) a large number of directions ($\approx 94\%$) have near-zero eigenvalues (magnitude less than 10^{-4}),

(ii) positive eigenvalues (right inset) have a long tail with the largest one being almost 40,

(iii) negative eigenvalues (left inset), which are directions of descent that the optimizer missed, have a much faster decay (the largest negative eigenvalue is only -0.46).

Non-convexity and optimization landscape in deep learning

- Interestingly, this trend is not unique to this particular network. Rather, its qualitative properties are shared across a variety of network architectures, network sizes, datasets or optimization algorithms.
- Local minima (this term is loosely used here, since a few of the Hessian eigenvalues may be slightly negative) that generalize well and are discovered by gradient descent lie in "wide valleys" of the energy landscape, rather than in sharp isolated minima.

Can we design optimization strategies that favor well-generalizable solutions lying in large flat regions of the energy landscape?

Non-convexity and optimization landscape in deep learning

Consider a cartoon energy landscape, where x-axis denotes the configuration space of the parameters, with two local minima: a shallower although wider one at x_{robust} and a very sharp global minimum at $x_{\text{non-robust}}$.

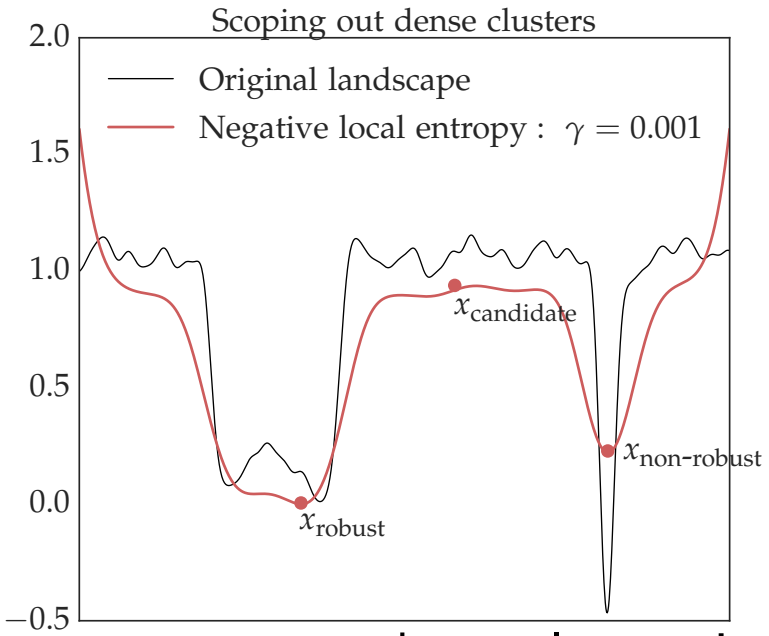


Figure: Local entropy concentrates on dense clusters in the energy landscape.

- neighborhood of x_{robust}: "local entropy" is large because it includes the contributions from a whole region of good configurations
- neighborhood of $x_{\text{non-robust}}$: there are no such contributions except for the minimum itself and the resulting local entropy is low

Local entropy - provides a way of picking large, approximately flat, regions of the landscape over sharp, narrow valleys in spite of the latter possibly having a lower loss.

Non-convexity and optimization landscape in deep learning

For a parameter vector $x \in \mathbb{R}^n$, consider a Gibbs distribution corresponding to a given energy landscape $f(x)$:

$$P(x;\beta) = Z_\beta^{-1} \exp(-\beta\ f(x)); \tag{1}$$

where β is known as the inverse temperature and Z_β is a normalizing constant, also known as the partition function in physics. As $\beta \to \infty$, the probability distribution above concentrates on the global minimum of $f(x)$ (assuming it's unique) given as:

$$x^* = \arg\min_x f(x), \tag{2}$$

which establishes the link between the Gibbs distribution and a generic optimization problem 2.

Non-convexity and optimization landscape in deep learning

We would instead like the probability distribution - and therefore the underlying optimization problem - to focus on flat regions such as x_{robust}, as in the figure. With this in mind, construct a modified Gibbs distribution:

$$P(x'; x, \beta, \gamma) = Z_{x,\beta,\gamma}^{-1} \exp\left(-\beta\ f(x') - \beta\frac{\gamma}{2}\|x - x'\|_2^2\right). \tag{3}$$

The distribution in (3) is a function of a dummy variable x' and is parameterized by the original location x.

The parameter γ biases the modified distribution (3) towards x:

- large γ results in a $P(x'; x, \beta, \gamma)$ with all its mass near x irrespective of the energy landscape of $f(x')$
- small values of γ: the term $f(x')$ in the exponent dominates and the modified distribution is similar to the original Gibbs distribution in (1).

We will set the inverse temperature β to 1 because γ affords us similar control on the Gibbs distribution.

Non-convexity and optimization landscape in deep learning

Definition 4 (Local entropy)

The local free entropy of the Gibbs distribution in (1), colloquially called "local entropy" will be denoted by $F(x, \gamma)$, is defined as the log-partition function of modified Gibbs distribution (3), i.e.,

$$\begin{aligned} F(x,\gamma) &= \log Z_{x,1,\gamma} \\ &= \log \int_{x'} \exp\left(-f(x') - \frac{\gamma}{2}\|x - x'\|_2^2\right) dx' \\ &= \log \int_{x'} \exp(-f(x')) \cdot \exp\left(-\frac{\gamma}{2}\|x - x'\|_2^2\right) dx' \end{aligned}$$

The parameter γ is used to focus the modified Gibbs distribution upon a local neighborhood of x and we call it a "scope".

Previous figure shows the negative local entropy $-F(x, \beta, \gamma)$ for the original energy landscape. Note that $-F(x, \beta, \gamma)$ has a global minimum near x_{robust} which is exactly what we want; indeed, x_{robust} has a higher local entropy than $x_{\text{non-robust}}$.

Non-convexity and optimization landscape in deep learning

We now present the **Entropy-SGD algorithm**, a variant of SGD that is motivated from local entropy.

Consider a classification setting, let $x \in \mathbb{R}^n$ be the weights of a deep neural network and $\xi_k \in \Xi$ be samples from a dataset Ξ of size N. Let $f(x; \xi_k)$ be the loss function, e.g., cross-entropy of the classifier on a sample ξ_k.

The original optimization problem is: $x^* = \arg\min_x \frac{1}{N} \sum_{k=1}^{N} f(x; \xi_k)$, where the objective $f(x, \xi_k)$ is typically, a non-convex function in both the weights x and the sample ξ_k. The Entropy-SGD algorithm instead solves

$$x^*_{\text{Entropy-SGD}} = \arg\min_x \rho \left(\frac{1}{N} \sum_{k=1}^{N} f(x; \xi_k) \right) - F(x, \gamma; \Xi); \tag{4}$$

where the dependence of local entropy $F(x, \gamma)$ on the dataset Ξ is explicit. ρ allows to control the contribution of the original back-propagated gradient in Entropy-SGD. In fact setting $\rho = 0$ (the modified loss function is then simply $-F(x, \gamma)$) is plausible and works well.

Entropy-SGD algorithm optimizes this objective and resembles two nested loops of SGD, where Langevin dynamics is used to compute the gradient of the local entropy at each update of the weights.