CENG501 – Deep Learning

Week 3

Spring 2026

Sinan Kalkan

Dept. of Computer Engineering, METU

Today

- Representational Capacity
- Overfitting, Convergence, When to Stop Training
- Data Preprocessing
- Weight Initialization
- Concluding Remarks
- CNNs

Administrative Notes

- ~~Reading assignment~~
  - ~~CH1-7 of the Hundred-Page Machine Learning Book by Andriy Burkov. https://themlbook.com/~~

- Quiz #1
  - Upload the PDF on ODTUclass.

- Paper Selection
  - https://forms.gle/SXm33dFc9GHnhWje8
  - Deadline next week (5th of March, midnight)

Representational capacity

Representational capacity

- Boolean functions:
  - Every Boolean function can be represented exactly by a neural network
  - The number of hidden layers might need to grow with the number of inputs
- Continuous functions:
  - Every bounded continuous function can be approximated with small error with two layers
- Arbitrary functions:
  - Three layers can approximate any arbitrary function

Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2 (4), 303-314

Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251257.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

Representational Capacity: Why go deeper if 3 layers is sufficient?

- Going deeper helps convergence in “big” problems.
- Going deeper in “old-fashion trained” ANNs does not help much in accuracy
  - However, with different training strategies or with Convolutional Networks, going deeper matters

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015). The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics* (pp. 192-204).

Representational Capacity

- More hidden neurons ➔ capacity to represent more complex functions

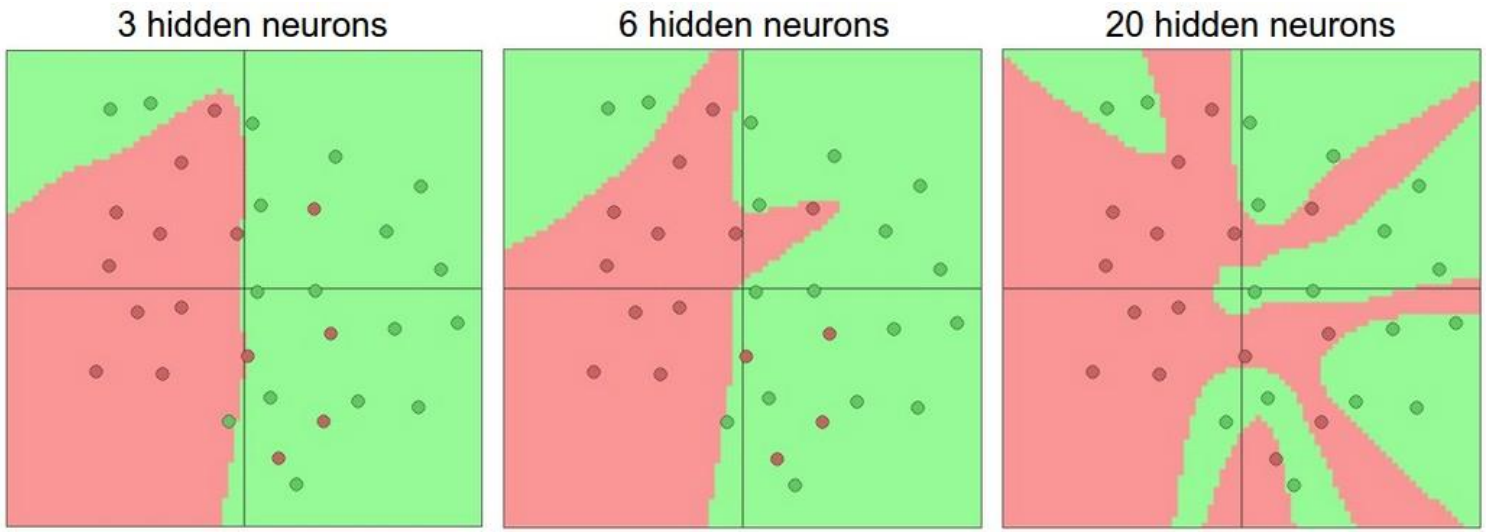


Figure: https://cs231n.github.io/

- Problem: overfitting vs. generalization
  - We will discuss the different strategies to help here (L2 regularization, dropout, input noise, using a validation set etc.)

Number of hidden neurons

Several rule of thumbs (Jeff Heaton)

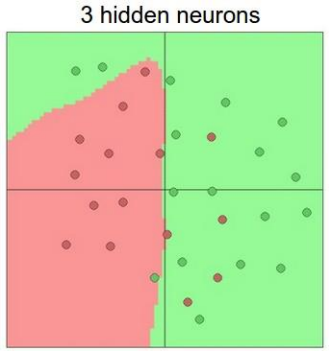
- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be:
  - 2/3 x (the size of the input layer + the size of the output layer)
- The number of hidden neurons should be less than twice the size of the input layer.

Number of hidden layers

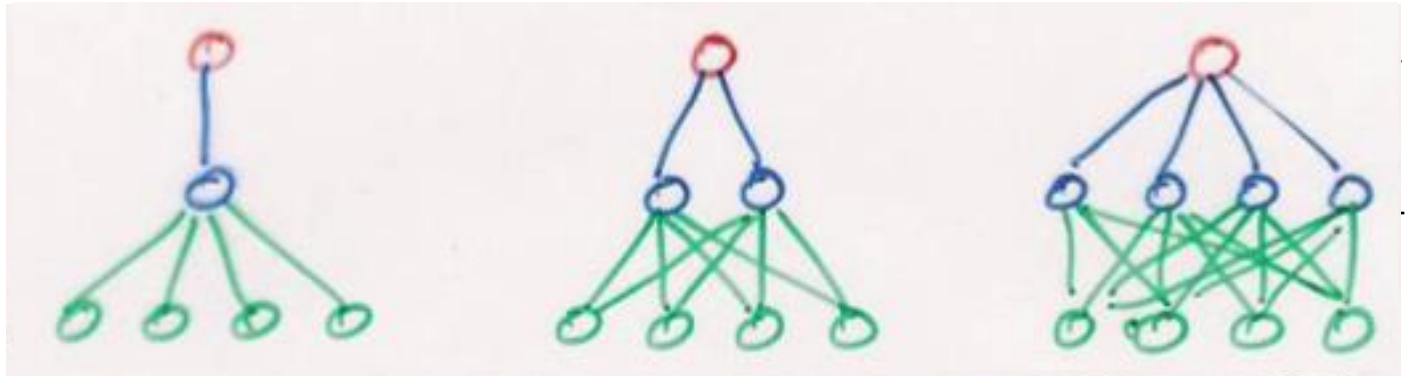
- Depends on the nature of the problem
  - Linear classification? ➔ No hidden layers needed
  - Non-linear classification?

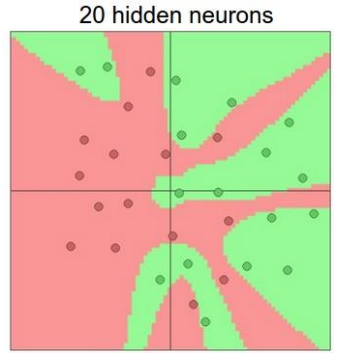
Model Complexity

- Models range in their flexibility to fit arbitrary data



https://cs231n.github.io/





https://cs231n.github.io/

high bias model

low variance

small capacity may prevent it from representing all structure in data

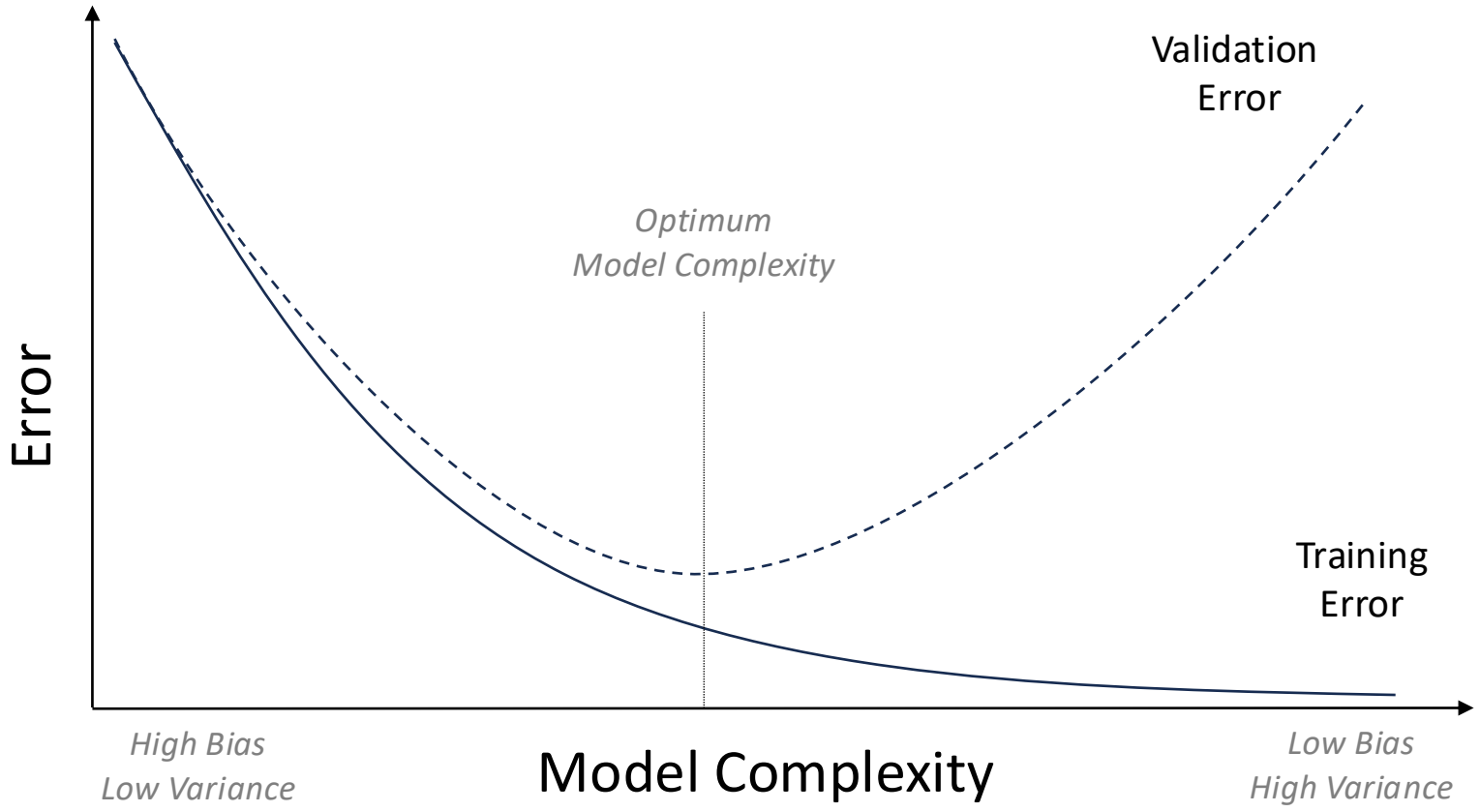
low bias model

high variance

large capacity may allow it to memorize data and fail to capture regularities

Slide Credit: Michael Mozer

Training Vs. Test/Val Set Error



Bias-Variance Trade Off

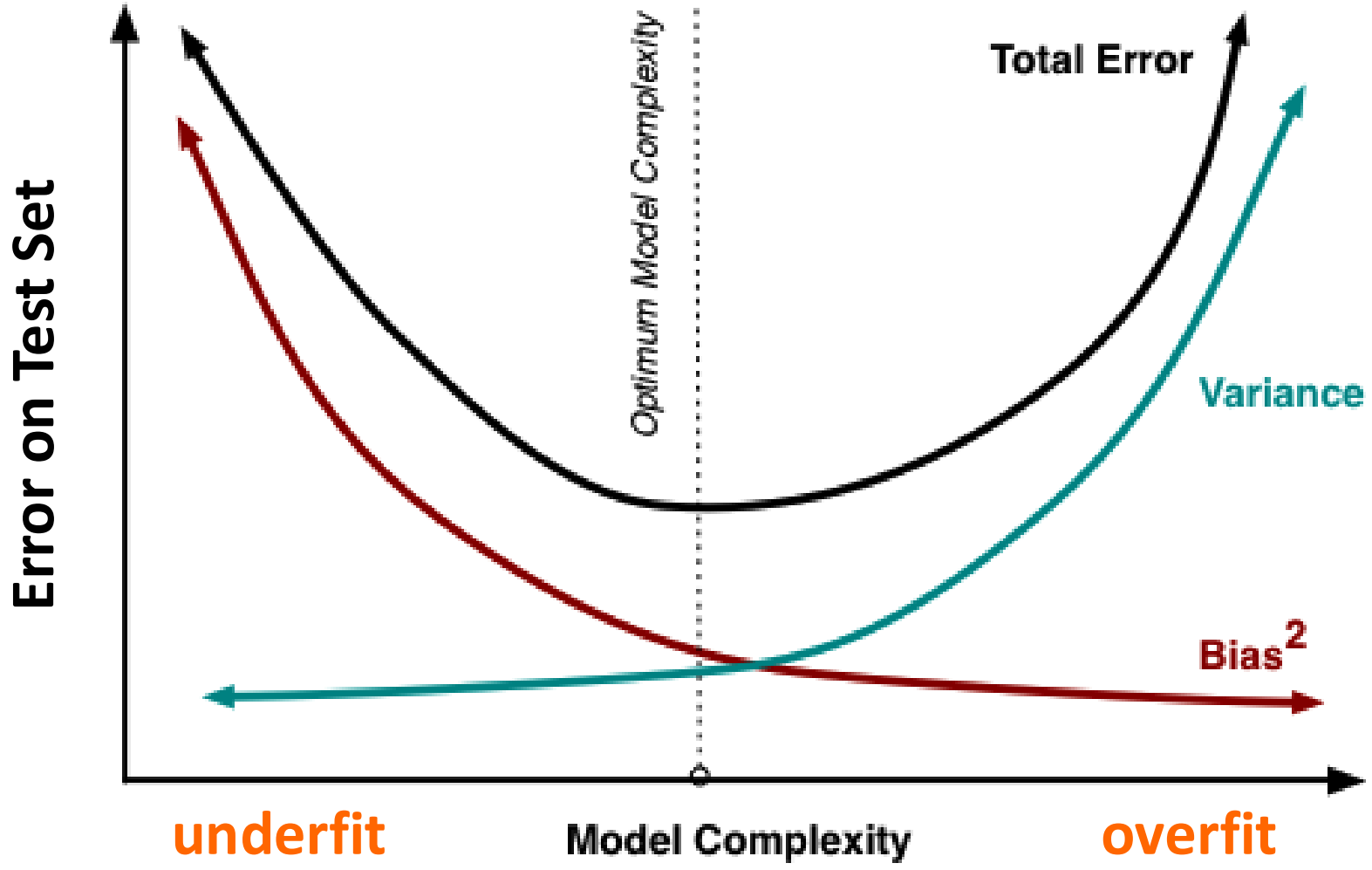


image credit: scott.fortmann-roe.com

Slide Credit: Michael Mozer

Memorization vs. Generalization

https://arxiv.org/pdf/1906.05271.pdf

Does Learning Require Memorization? A Short Tale about a Long Tail*

Vitaly Feldman
Google Research†
Mountain View, CA, USA
vitaly.edu@gmail.com

ABSTRACT

State-of-the-art results on image recognition tasks are achieved using over-parameterized learning algorithms that (nearly) perfectly fit the training set and are known to fit well even random labels. This tendency to memorize seemingly useless training data labels is not explained by existing theoretical analyses. Memorization of the training data also presents significant privacy risks when the training data contains sensitive personal information and thus it is important to understand whether such memorization is necessary for accurate learning.

We provide a simple conceptual explanation and a theoretical model demonstrating that for natural data distributions memorization of labels is *necessary* for achieving close-to-optimal generalization error. The model is motivated and supported by the results of several recent empirical works. In our model, data is sampled from a mixture of subpopulations and the frequencies of these subpopulations are chosen from some prior. The model allows to quantify the effect of not fitting the training data on the generalization performance of the learned classifier and demonstrates that memorization is necessary whenever frequencies are long-tailed. Image and text data are known to follow such distributions and therefore our results establish a formal link between these empirical phenomena. Our results also have concrete implications for the cost of ensuring differential privacy in learning.

ACM Reference Format:
Vitaly Feldman. 2020. Does Learning Require Memorization? A Short Tale about a Long Tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3357713.3384290

1 INTRODUCTION

Understanding the generalization properties of learning systems based on deep neural networks (DNNs) is an area of great practical importance and significant theoretical interest. The main conceptual hurdle to adapting the classical approaches for analysis of generalization is the well-known fact that state-of-the-art approaches to training DNNs reach zero (or very low) training error even when the test error is relatively high. In fact, as highlighted in the influential work of Zhang *et al.*[54], low training error is achieved even when the labels are generated at random. The only way to fit an example whose label cannot be predicted based on the rest of the dataset is to effectively memorize it. In this work we will formalize and quantify this notion of memorization. For now we will informally say that a learning algorithm memorizes the label of some example (x, y) in its dataset S if the model output on S predicts y on x whereas when the learning algorithm is trained on S without (x, y) it is unlikely to predict y on x.

The classical approach to understanding generalization starts with the decomposition of the generalization error $\mathtt{err}_P(h)$ relative

Double Descent

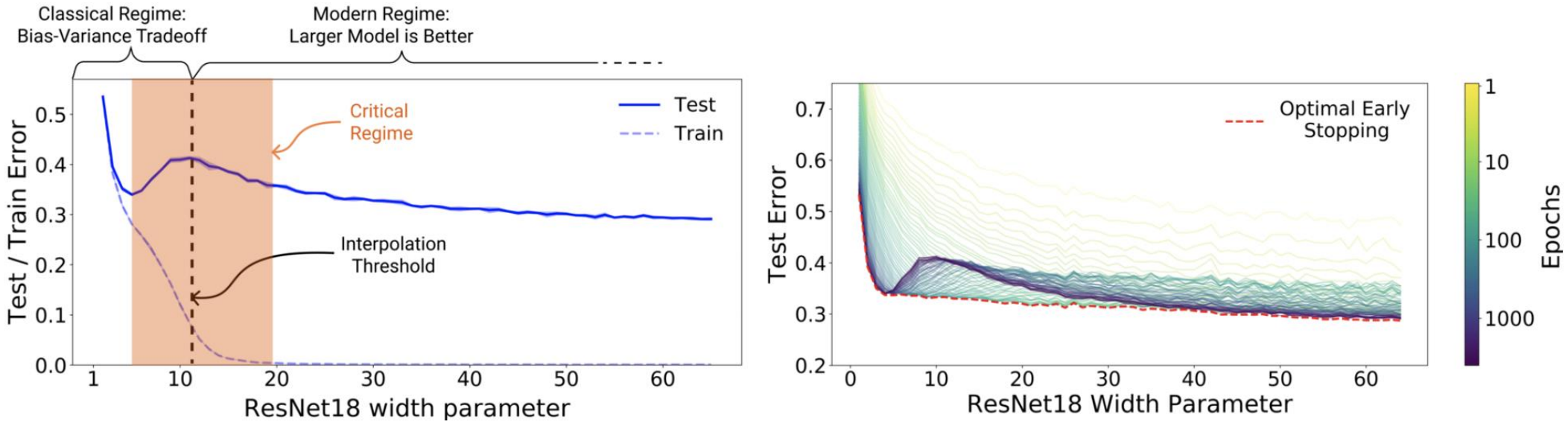
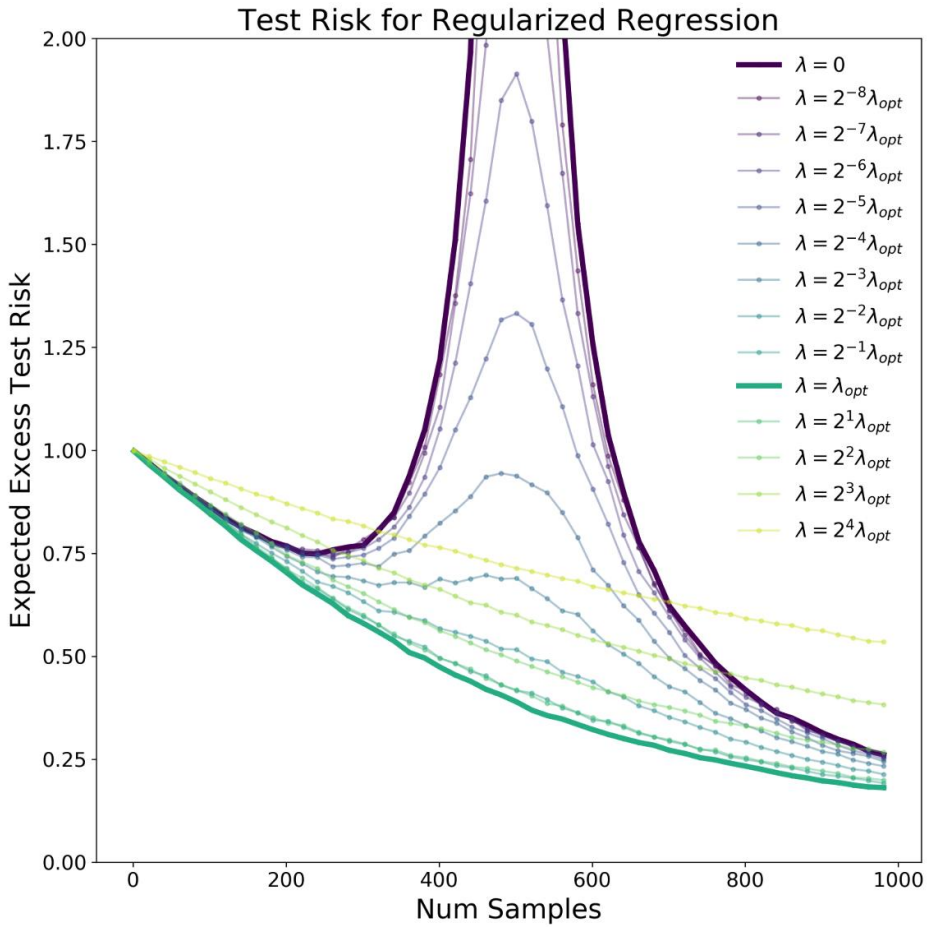


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

Nakkiran et al., "Deep Double Descent: Where Bigger Models and More Data Hurt", 2019.

Double Descent



Nakkiran et al., “Optimal Regularization Can Mitigate Double Descent”, 2020.

Grokking

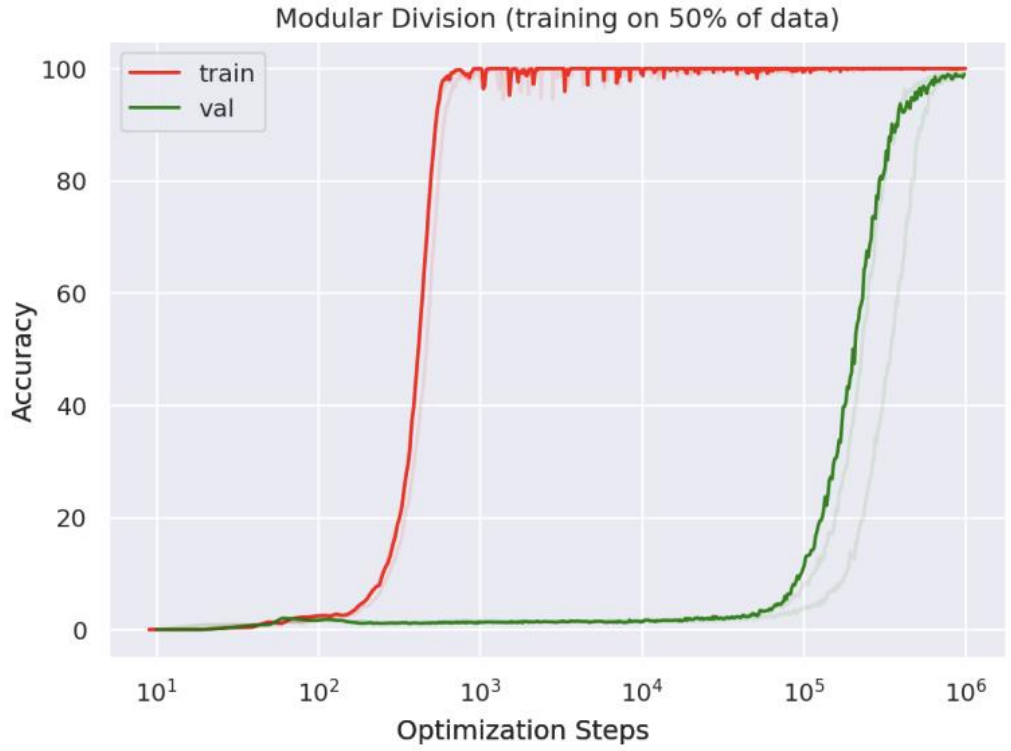
GROKKING: GENERALIZATION BEYOND OVERFITTING ON SMALL ALGORITHMIC DATASETS

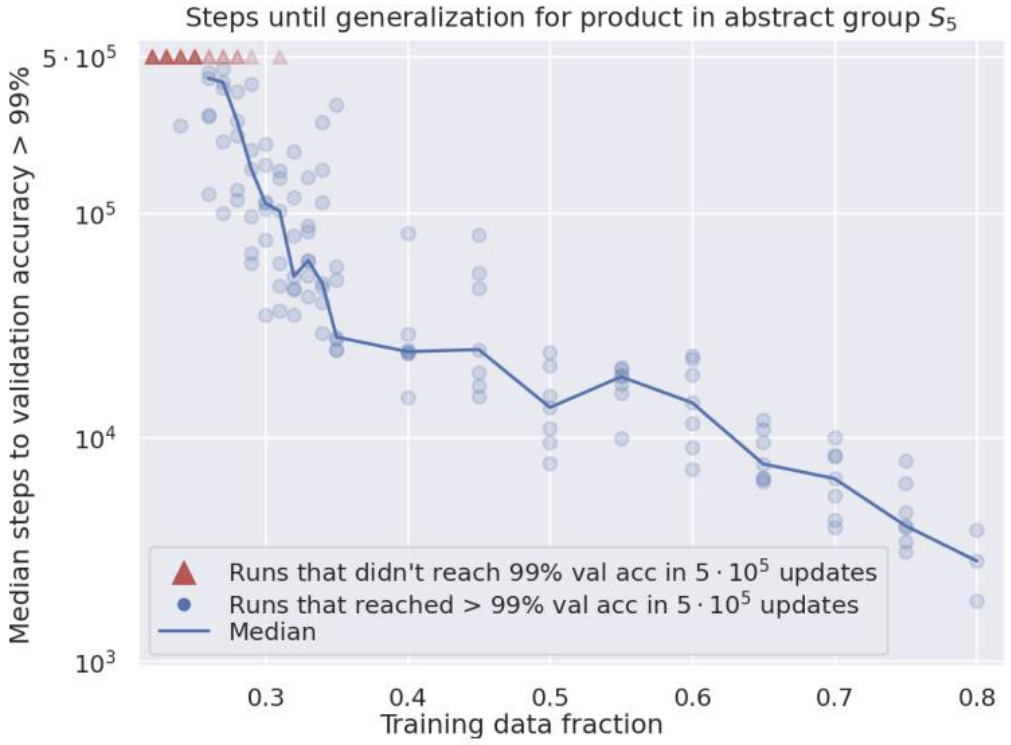
Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin
OpenAI

Vedant Misra*
Google

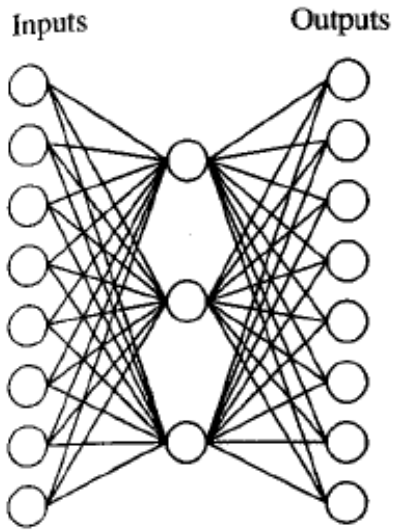
ABSTRACT

In this paper we propose to study generalization of neural networks on small algorithmically generated datasets. In this setting, questions about data efficiency, memorization, generalization, and speed of learning can be studied in great detail. In some situations we show that neural networks learn through a process of "grokking" a pattern in the data, improving generalization performance from random chance level to perfect generalization, and that this improvement in generalization can happen well past the point of overfitting. We also study generalization as a function of dataset size and find that smaller datasets require increasing amounts of optimization for generalization. We argue that these datasets provide a fertile ground for studying a poorly understood aspect of deep learning: generalization of overparametrized neural networks beyond memorization of the finite training dataset.





What do the layers represent?

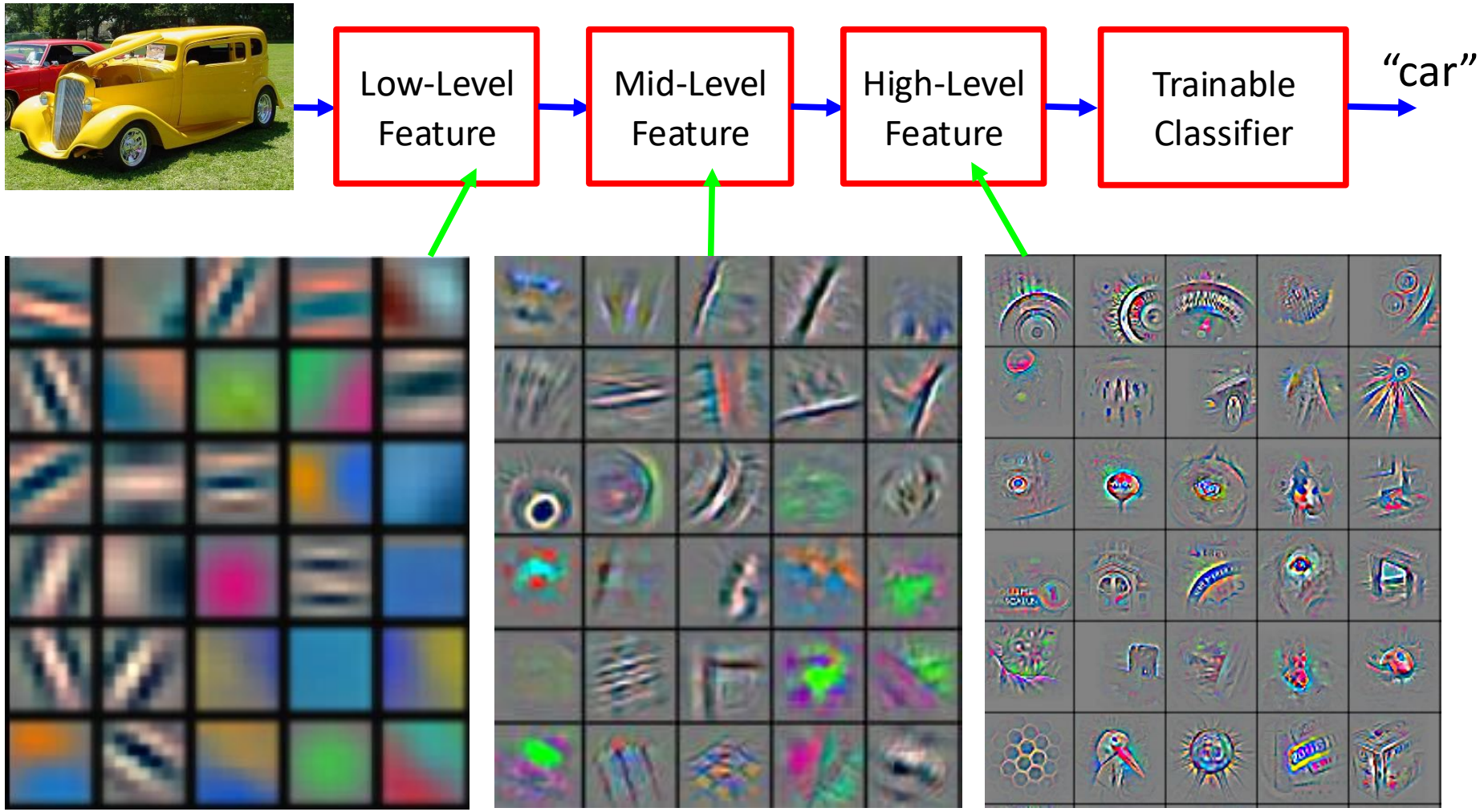


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

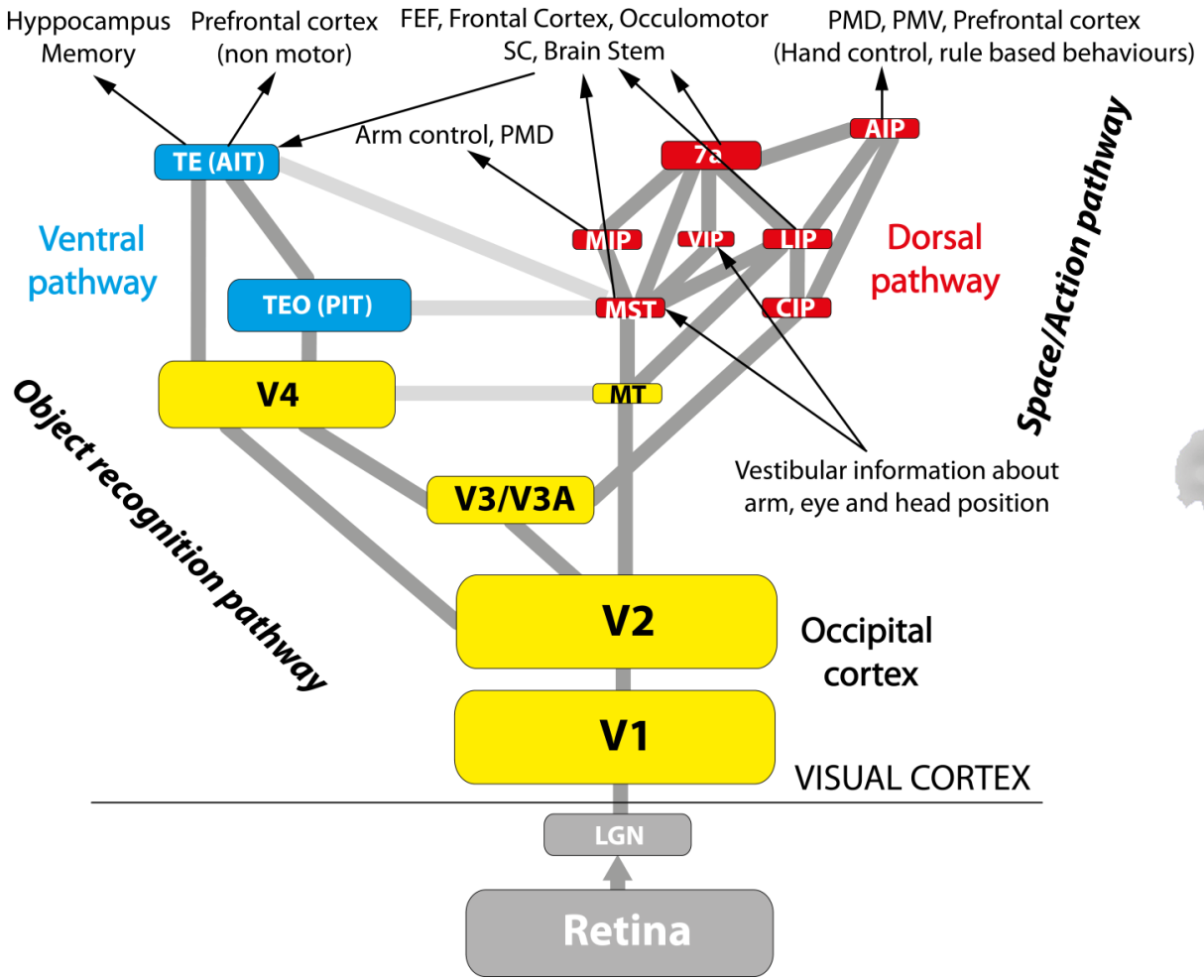
T. Mitchell, “Machine Learning”, 1997.

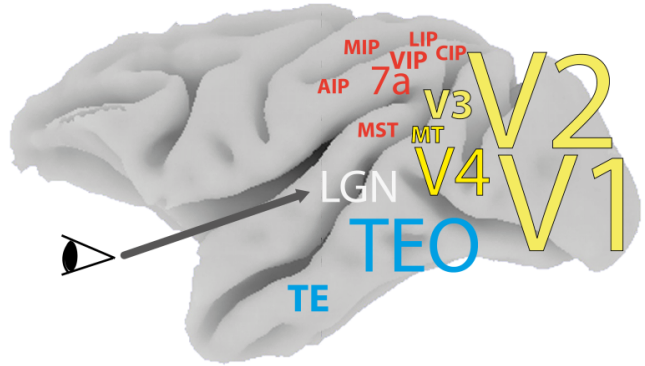
What do the layers represent?



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Similarities to the Hierarchies in Visual Cortex





Krueger, Jannsen, Kalkan, Lappe, .., “Deep Hierarchies in the Primate Visual Cortex: What Can We Learn For Computer Vision”, IEEE PAMI, 2013.

Overfitting, Convergence, and when to stop

Overfitting

- Occurs when training procedure fits to not only regularities in training data but also noise.
  - Like memorizing the training examples instead of learning the statistical regularities
- Leads to poor performance on test set
- Most of the practical issues with neural nets involve avoiding overfitting

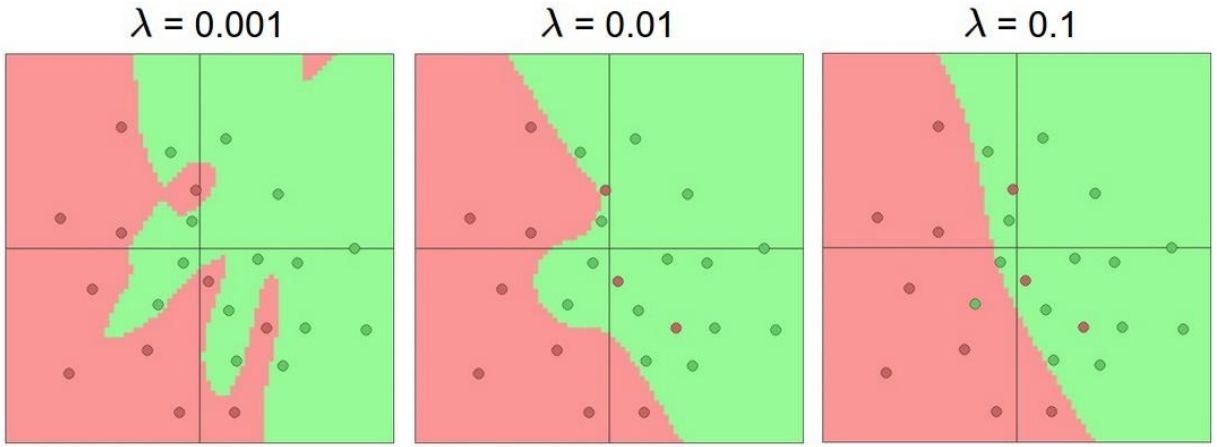
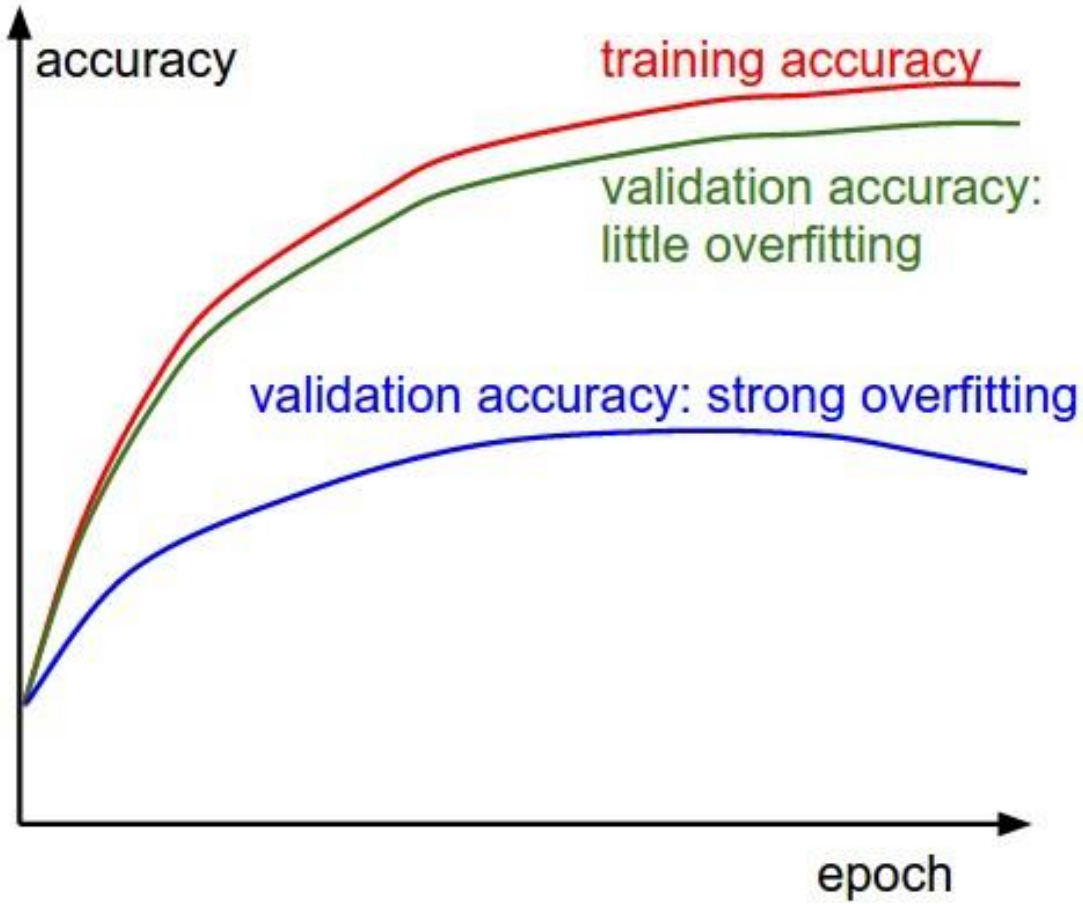


Figure: https://cs231n.github.io/

Adapted from Michael Mozer

How do you spot overfitting?



Avoiding Overfitting

- Increase training set size
  - Make sure effective size is growing; redundancy doesn't help
- Incorporate domain-appropriate bias into model
  - Customize model to your problem
- Tune hyperparameters of model
  - number of layers, number of hidden units per layer, connectivity, etc.
- Regularization techniques

Slide Credit: Michael Mozer

Incorporating Domain-Appropriate Bias Into Model

- Input representation
- Output representation
- Architecture
  - # layers, connectivity
  - e.g., convolutional nets, residual connections etc.
- Activation function
- Loss function

Slide Credit: Michael Mozer

Customizing Networks

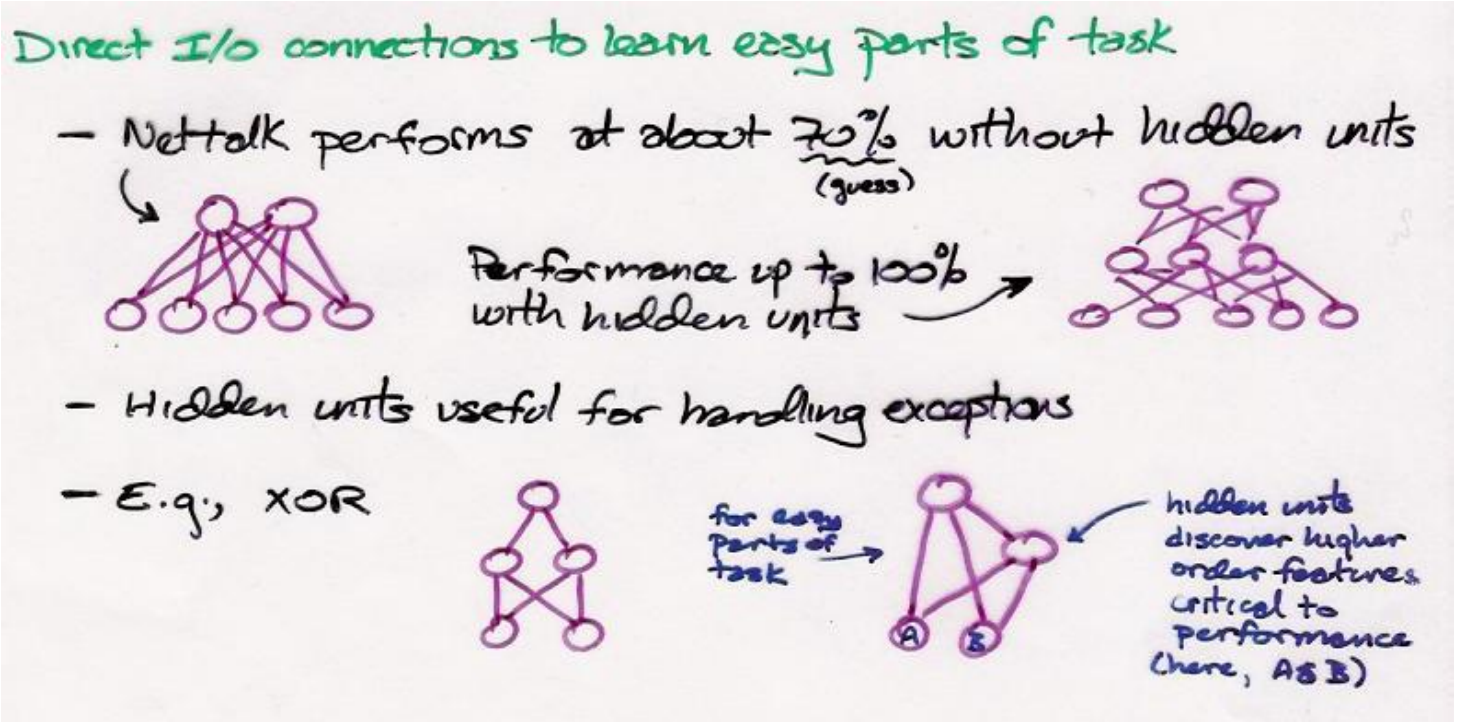
- Neural nets can be customized based on the problem domain
  - choice of loss function
  - choice of activation function

- Domain knowledge can be used to impose domain-appropriate bias on model
  - bias is good if it reflects properties of the data set
  - bias is harmful if it conflicts with properties of data

Slide Credit: Michael Mozer

Adding bias into a model

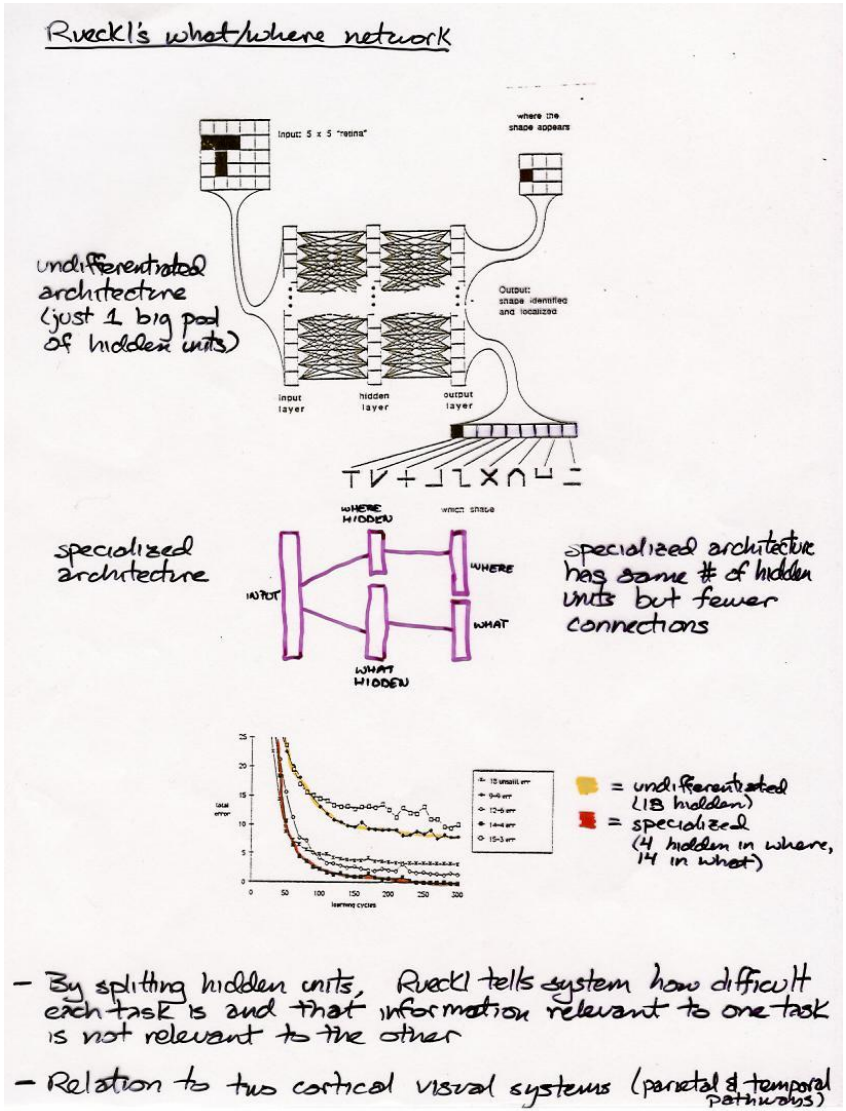
- Adding hidden layers or direct connections based on the problem



Slide Credit: Michael Mozer

Adding bias into a model

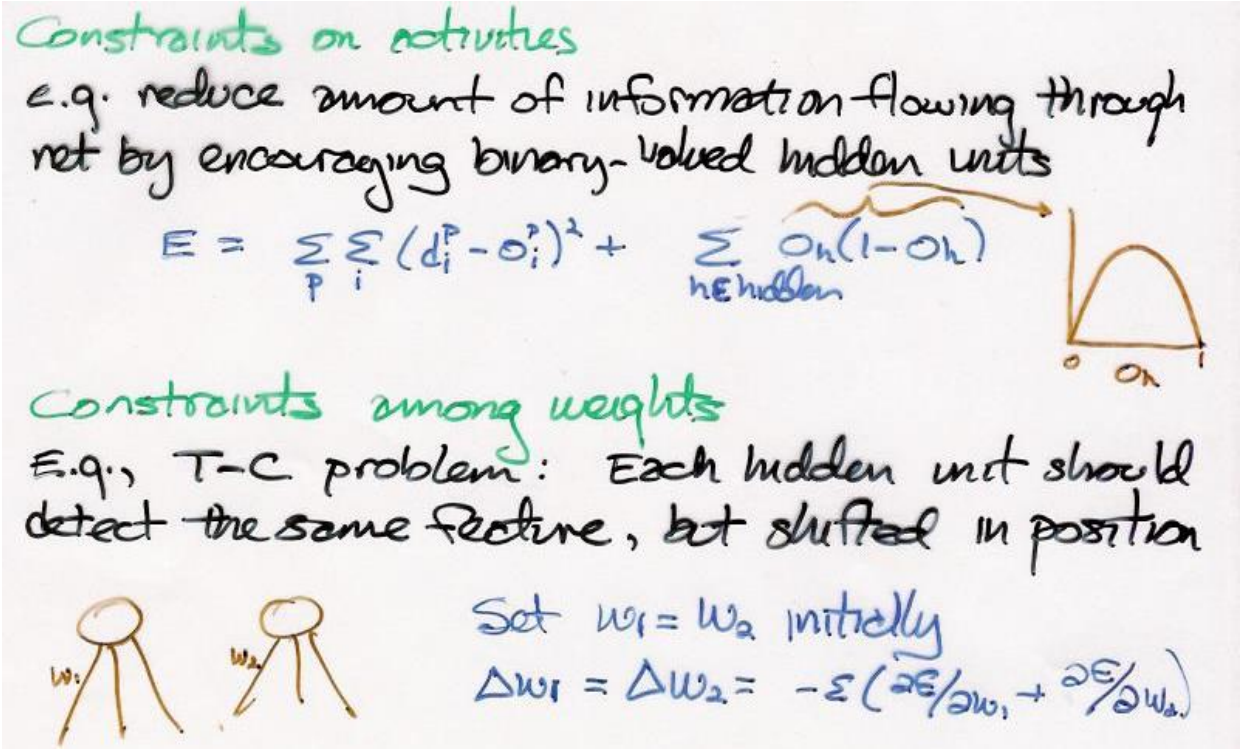
- Modular architectures
  - Specialized hidden units for special problems



Slide Credit: Michael Mozer

Adding bias into a model

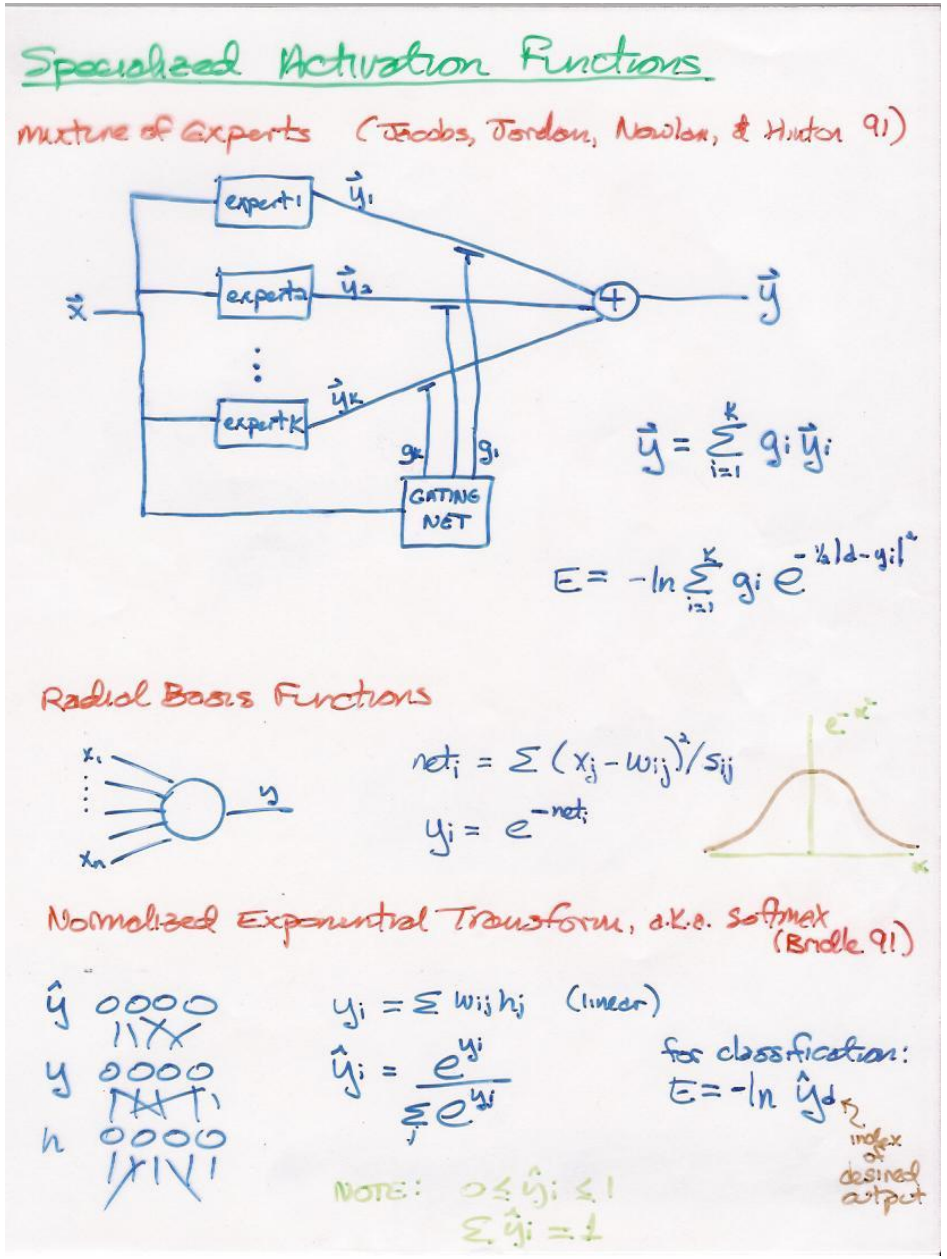
- Local or specialized receptive fields
  - E.g., in CNNs
- Constraints on activities
- Constraints on weights



Slide Credit: Michael Mozer

Adding bias into a model

- Use different loss functions (e.g., cross-entropy)
- Use specialized activation functions



Slide Credit: Michael Mozer

Adding bias into a model

- Introduce other parameters
  - Temperature
  - Saliency of input

Designing bias into the net (contd.)

Introduce parameters other than weights/biases and perform gradient descent in these parameters as well.

– E.g., "temperature" (steepness of sigmoid)

$$o_i = \frac{1}{1+e^{-net_i/T_i}}$$

Compute $\partial E/\partial T_i$ $\quad \Delta T_i = -\epsilon \frac{\partial E}{\partial T_i}$

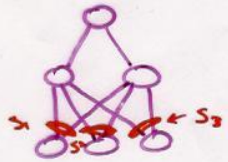
– E.g., input salience term

In the "real world" many inputs are irrelevant to task at hand. Would like to suppress them.

$$net_i \text{ (layer 2)} = \sum w_{ij} o_j s_j$$

o_j: activity of unit j in layer 1

s_j: salience of unit j in layer 1 ($0-1$)



Compute $\partial E/\partial s_j$ $\quad \Delta s_j = -\epsilon \, \partial E/\partial s_j$

Equivalent to changing all outgoing weights from input unit simultaneously

– These parameters allow you to cut across weight space diagonally (low T ≡ turning up all weights coming into unit; low S ≡ turning down all weights coming from unit)

Slide Credit: Michael Mozer

Regularization

- Regularization strength can effect overfitting

$$\frac{1}{2}\lambda w^2$$

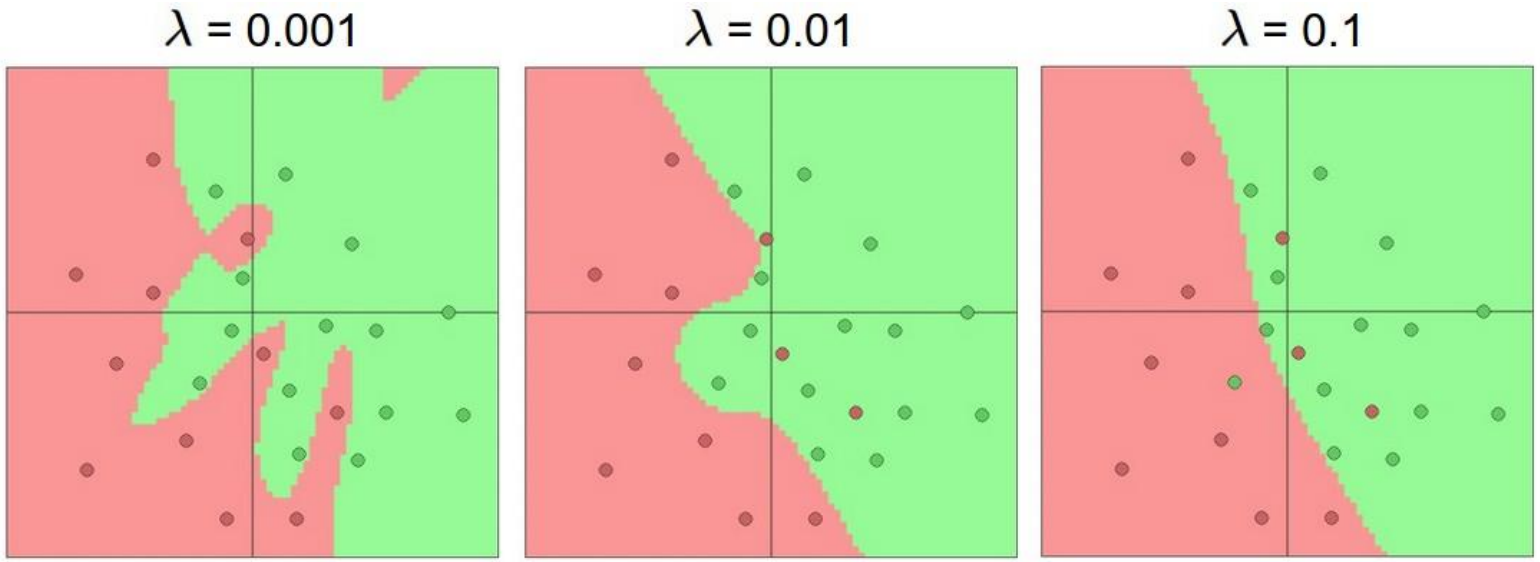


Figure: https://cs231n.github.io/

Regularization

- L2 regularization: $\frac{1}{2}\lambda w^2$
  - Very common
  - Penalizes peaky weight vector, prefers diffuse weight vectors

- L1 regularization: $\lambda|w|$
  - Enforces sparsity (some weights become zero)
  - Why? Weight decay is by a constant value if |w| is non-zero.
  - Leads to input selection (makes it noise robust)
  - Use it if you require sparsity / feature selection

- Can be combined: $\lambda_1|w| + \lambda_2 w^2$

- Regularization is not performed on the bias; it seems to make no significant difference

L2 regularization and weight decay

$$L = L_{data} + \frac{1}{2}\lambda w^2$$

- L2 regularization

$$w_i \leftarrow w_i - \eta\left(\frac{\partial L_{data}}{\partial w_i} + \lambda w_i\right)$$

$$\Delta w_i \leftarrow \mu\,\Delta w_i + (1-\mu)\left(\frac{\partial L_{data}}{\partial w_i} + \lambda w_i\right)$$

$$w_i \leftarrow w_i - \eta\,\Delta w_i$$

When you add moving avg (as in e.g. Adam), they become different

Vs.

- Weight decay

$$w_i \leftarrow w_i - \eta\frac{\partial L_{data}}{\partial w_i} - \eta\lambda w_i$$

$$\Delta w_i \leftarrow \mu\,\Delta w_i + (1-\mu)\left(\frac{\partial L_{data}}{\partial w_i}\right)$$

$$w_i \leftarrow w_i - \eta\Delta w_i - \eta\lambda w_i$$

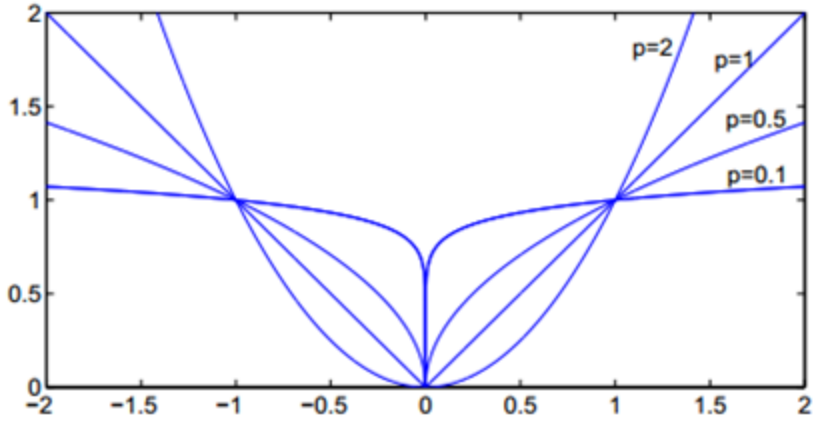
Weight Decay

- Adam & weight decay issue:

https://www.fast.ai/2018/07/02/adam-weight-decay/

L0 regularization

- $L_0 = \left(\sum_i x_i^0\right)^{1/0}$
- How to compute the zeroth power and zeroth-root?
- Mathematicians approximate this as:
  - $L_0 = \#\{i \mid x_i \neq 0\}$
  - The cardinality of non-zero elements
- This is a strong enforcement of sparsity.
- However, this is non-convex
  - L1 norm is the closest convex form



Probabilistic interpretation of regularization

- http://bjlkeng.github.io/posts/probabilistic-interpretation-of-regularization/

- https://towardsdatascience.com/understanding-the-scaling-of-l%C2%B2-regularization-in-the-context-of-neural-networks-e3d25f8b50db

- Adverse effects of regularization and normalization:

https://ojs.aaai.org/index.php/AAAI/article/view/6046

https://arxiv.org/abs/1911.05920

Large-norm L2 regularization

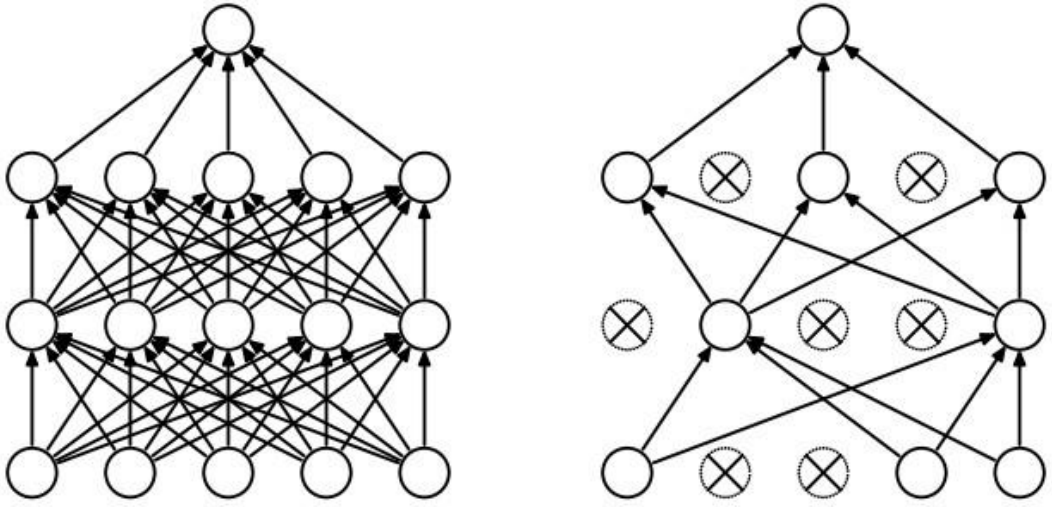
- https://arxiv.org/pdf/1910.00359.pdf
- (section 3)

Regularization

- Enforce an upper bound on weights:
  - Max norm:
    - $\left|\left|w\right|\right|_2 < c$
    - Helps the gradient explosion problem
    - Improvements reported

- Dropout:
  - At each iteration, *drop* a number of neurons in the network
  - **Use a neuron's activation** with probability p (a hyperparameter)
  - Adds stochasticity!

Fig: Srivastava et al., 2014



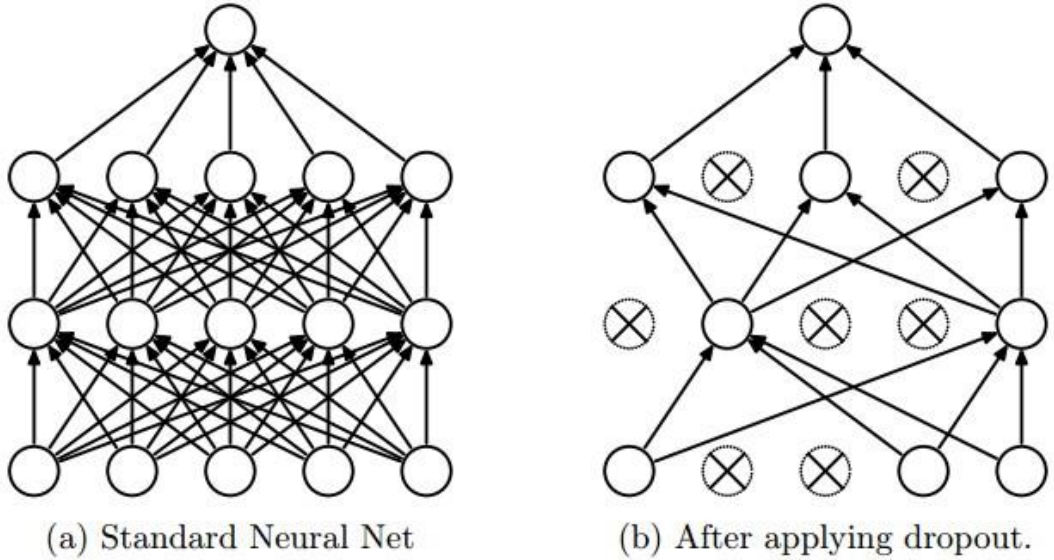
(a) Standard Neural Net

(b) After applying dropout.

http://cs231n.github.io/neural-networks-2/

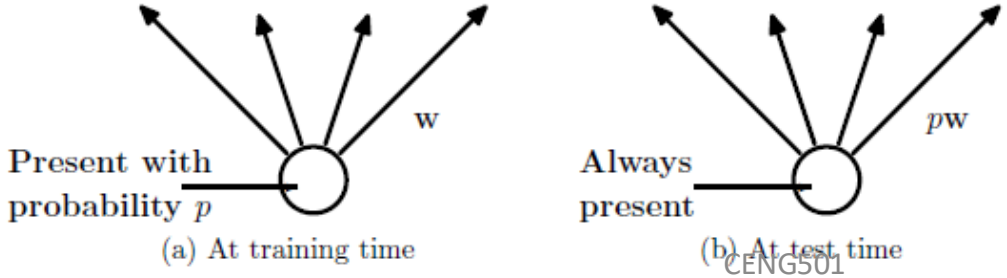
Regularization: Dropout

Fig: Srivastava et al., 2014



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

- Feed-forward only on active units
- Can be trained using SGD with mini-batch
  - Back propagate only “active” units.
- One issue:
  - Expected output x with dropout:
  - $E[x'] = \frac{1}{N}\sum_i (px_i + (1-p)0) = p\frac{1}{N}\sum_i x_i = pE[x]$
- To have the same scale at testing time (no dropout), multiply test-time activations with p.



(a) At training time (b) At test time

Fig: Srivastava et al., 2014

Regularization: Dropout

Training-time:

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = np.random.rand(*H1.shape) < p # first dropout mask
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = np.random.rand(*H2.shape) < p # second dropout mask
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

Test-time: All neurons receive their normal input (x) so we should scale by p to have $E[x] = px$.

```
# ensembled forward pass
H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
out = np.dot(W3, H2) + b3
```


http://cs231n.github.io/neural-networks-2/

Regularization: Inverted Dropout

Perform scaling while dropping at training time!

Training-time: Correct the expected expected output from px to x.

```
# forward pass for example 3-layer neural network
H1 = np.maximum(0, np.dot(W1, X) + b1)
U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
H1 *= U1 # drop!
H2 = np.maximum(0, np.dot(W2, H1) + b2)
U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
H2 *= U2 # drop!
out = np.dot(W3, H2) + b3
```

Test-time:

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```


http://cs231n.github.io/neural-networks-2/

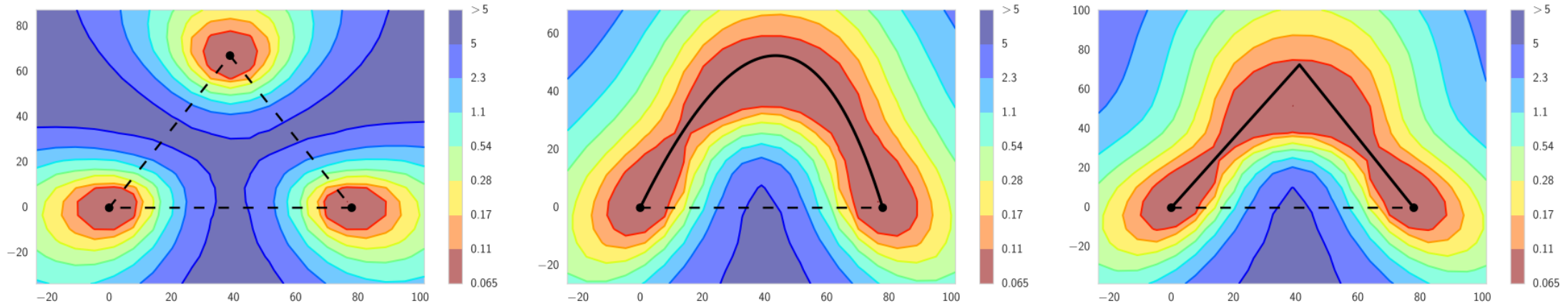
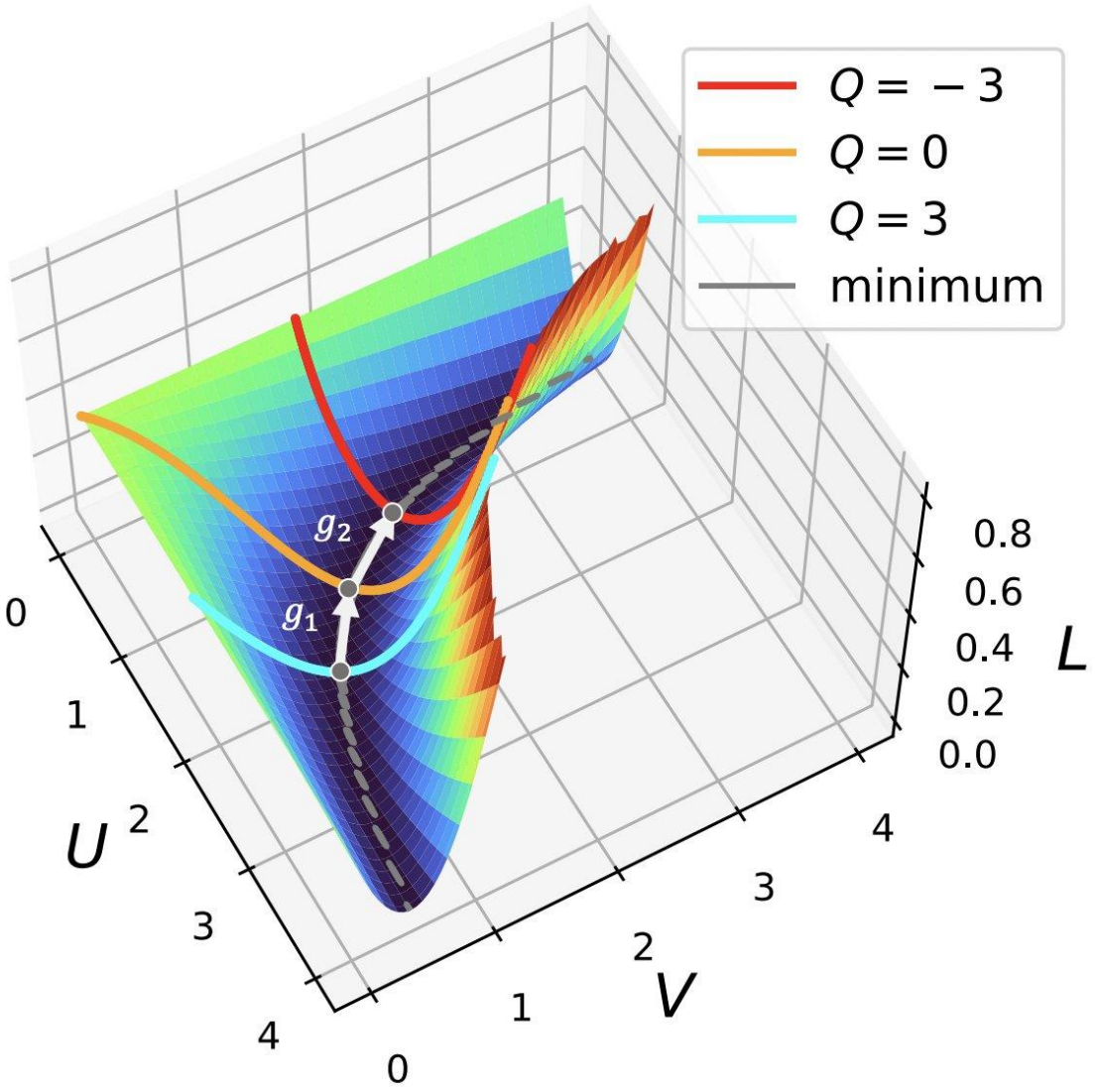


Figure 1: The ℓ_2-regularized cross-entropy train loss surface of a ResNet-164 on CIFAR-100, as a function of network weights in a two-dimensional subspace. In each panel, the horizontal axis is fixed and is attached to the optima of two independently trained networks. The vertical axis changes between panels as we change planes (defined in the main text). **Left:** Three optima for independently trained networks. **Middle** and **Right**: A quadratic Bezier curve, and a polygonal chain with one bend, connecting the lower two optima on the left panel along a path of near-constant loss. Notice that in each panel a direct linear path between each mode would incur high loss.

Garipov et al., “Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs”, 2018.

See also: Kuditipudi et al., “Explaining Landscape Connectivity of Low-cost Solutions for Multilayer Nets”, 2020.
- Explains this with noise stability, dropout stability.



SYMMETRIES, FLAT MINIMA AND THE CONSERVED QUANTITIES OF GRADIENT FLOW

Bo Zhao[*†]
University of California, San Diego
bozhao@ucsd.edu

Iordan Ganev[*]
Radboud University
iganev@cs.ru.nl

Robin Walters
Northeastern University
r.walters@northeastern.edu

Rose Yu
University of California, San Diego
roseyu@ucsd.edu

Nima Dehmamy
IBM Research
nima.dehmamy@ibm.com

ABSTRACT

Empirical studies of the loss landscape of deep networks have revealed that many local minima are connected through low-loss valleys. Yet, little is known about the theoretical origin of such valleys. We present a general framework for finding continuous symmetries in the parameter space, which carve out low-loss valleys. Our framework uses equivariances of the activation functions and can be applied to different layer architectures. To generalize this framework to nonlinear neural networks, we introduce a novel set of nonlinear, data-dependent symmetries. These symmetries can transform a trained model such that it performs similarly on new samples, which allows ensemble building that improves robustness under certain adversarial attacks. We then show that conserved quantities associated with linear symmetries can be used to define coordinates along low-loss valleys. The conserved quantities help reveal that using common initialization methods, gradient flow only explores a small part of the global minimum. By relating conserved quantities to convergence rate and sharpness of the minimum, we provide insights on how initialization impacts convergence and generalizability.

https://openreview.net/pdf?id=9ZpciCOunFb

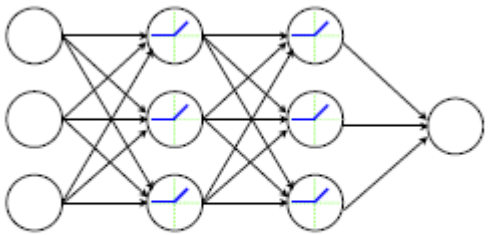
Drop-Activation: Implicit Parameter Reduction and Harmonic Regularization

Senwei Liang
National University of Singapore
10 Lower Kent Ridge Road
Singapore 119076
liangsenwei@u.nus.edu

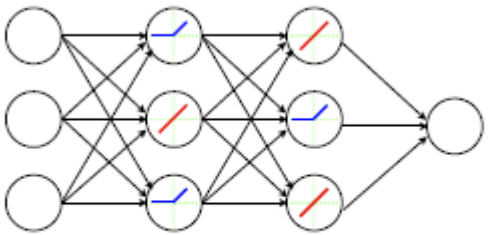
Yuehaw Kwoo
Stanford University
450 Serra Mall
Stanford, CA 94305
ykhoo@stanford.edu

Haizhao Yang
National University of Singapore
10 Lower Kent Ridge Road
Singapore 119076
haizhao@nus.edu.sg

14 November 2018



(a) Standard neural network with nonlinearity



(b) After applying Drop-Activation

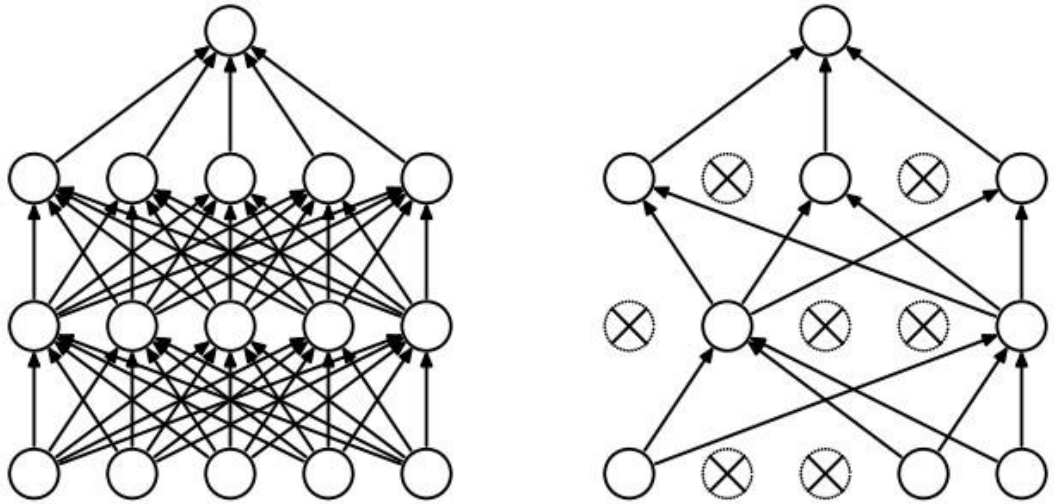
model	Baseline	DropAct
ResNet-164	8.85	**8.82**
PreResNet-164	8.88	**8.72**
WideResNet-28-10	8.97	**8.72**
DenseNet-BC-100-12	**8.81**	8.90
ResNeXt-29-8x64d	9.07	**8.91**

Table 4: Test error (%) on EMNIST (Balanced). The Baseline results were generated by ourselves.

Dropout as Ensemble Training Method

"Dropout performs gradient descent on-line with respect to both the training examples and the ensemble of all possible subnetworks."

Fig: Srivastava et al., 2014



(a) Standard Neural Net

(b) After applying dropout.

Pierre Baldi and Peter J Sadowski. Understanding dropout. In Advances in neural information processing systems, pp. 2814–2822, 2013.

Dropout is a special case of the stochastic delta rule: faster and more accurate deep learning

Noah Frazier-Logue
Rutgers University Brain Imaging Center
Rutgers University - Newark
Newark, NJ 07103
n.frazier.logue@nyu.edu

Stephen José Hanson
Rutgers University Brain Imaging Center
Rutgers University - Newark
Newark, NJ 07103
jose@rubic.rutgers.edu

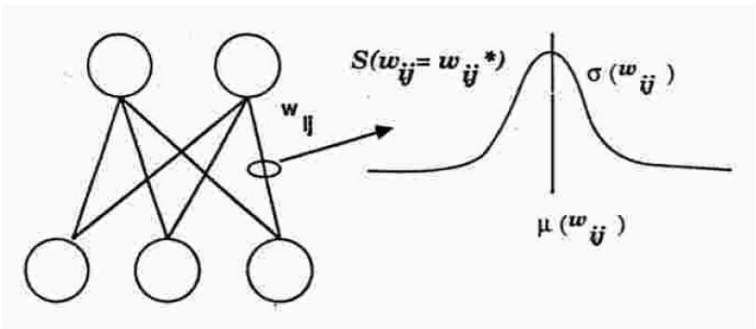


Figure 1: SDR sampling.

$$S(w_{ij} = w_{ij}^*) = \mu_{w_{ij}} + \mu_{w_{ij}}\theta(w_{ij}; 0, 1)$$

The first update rule refers to the mean of the weight distribution:

$$\mu_{w_{ij}}(n+1) = \alpha(\frac{\partial E}{\partial w_{ij}^*}) + \mu_{w_{ij}}(n)$$

and is directly dependent on the error gradient and has learning rate α. This is the usual delta rule update but conditioned on sample weights thus causing weight sharing through the updated mean value. The second update rule is for the standard deviation of the weight distribution (and for a Gaussian is known to be sufficient for identification).

$$\sigma_{w_{ij}}(n+1) = \beta|\frac{\partial E}{\partial w_{ij}^*}| + \sigma_{w_{ij}}(n)$$

Lottery Ticket Hypothesis

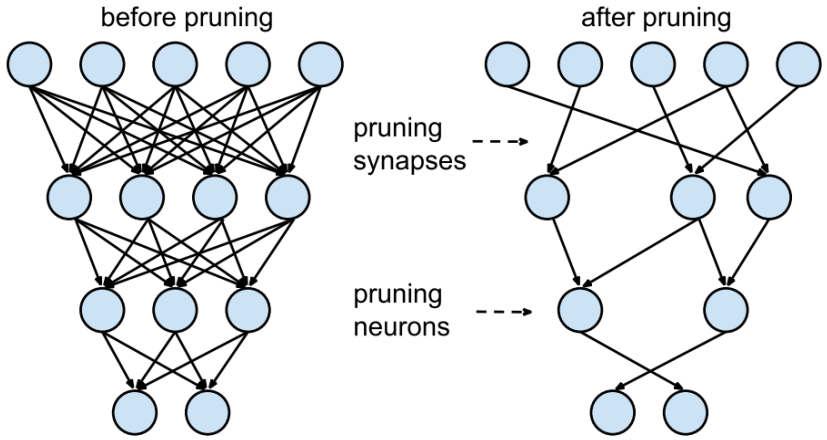


Figure: https://herbiebradley.com/The-Lottery-Ticket-Hypothesis

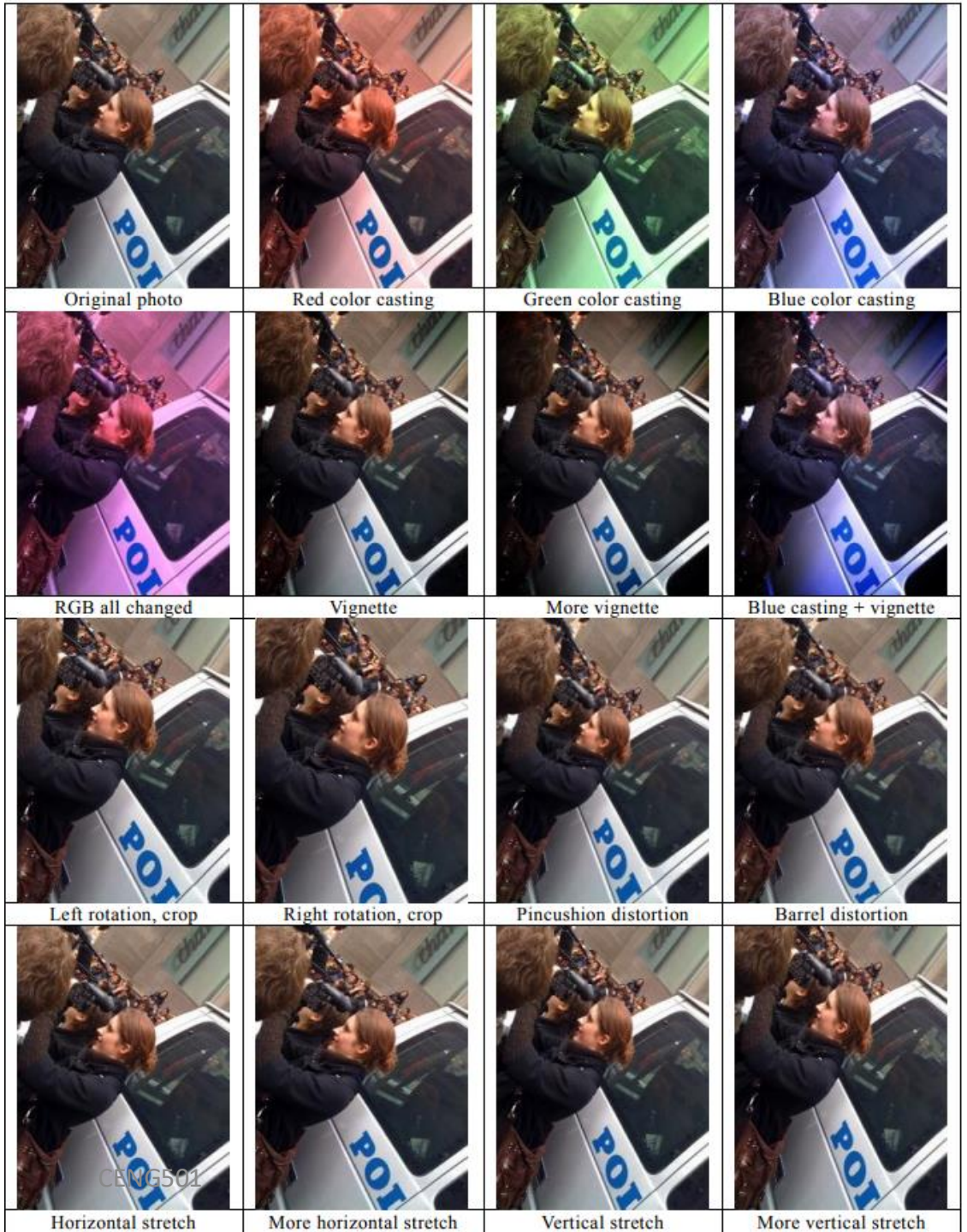
Frankle & Carbin, "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural networks", 2019.

Identifying winning tickets. We identify a winning ticket by training a network and pruning its smallest-magnitude weights. The remaining, unpruned connections constitute the architecture of the winning ticket. Unique to our work, each unpruned connection's value is then reset to its initialization from original network *before* it was trained. This forms our central experiment:

1. Randomly initialize a neural network $f(x;\theta_0)$ (where $\theta_0 \sim \mathcal{D}_\theta$).
2. Train the network for j iterations, arriving at parameters θ_j.
3. Prune $p\%$ of the parameters in θ_j, creating a mask m.
4. Reset the remaining parameters to their values in θ_0, creating the winning ticket $f(x; m \odot \theta_0)$.

As described, this pruning approach is *one-shot*: the network is trained once, $p\%$ of weights are pruned, and the surviving weights are reset. However, in this paper, we focus on *iterative pruning*, which repeatedly trains, prunes, and resets the network over n rounds; each round prunes $p^{\frac{1}{n}}\%$ of the weights that survive the previous round. Our results show that iterative pruning finds winning tickets that match the accuracy of the original network at smaller sizes than does one-shot pruning.

Data Augmentation



http://blcv.pl/static//2018/02/27/demystifying-face-recognition-v-data-augmentation/

Regularization Summary

- L2 regularization
- Inverted dropout with $p = 0.5$ (tunable)
- Data augmentation

When To Stop Training

- 1. Train *n* epochs; lower learning rate; train *m* epochs
  - bad idea: can't assume one-size-fits-all approach
- 2. Loss-change criterion
  - stop when loss isn't dropping
  - recommendation: criterion based on % drop over a window of, say, 10 epochs
    - 1 epoch is too noisy
    - absolute error criterion is too problem dependent
  - Another idea: train for a fixed number of epochs after criterion is reached (possibly with lower learning rate)

Slide Credit: Michael Mozer

When To Stop Training

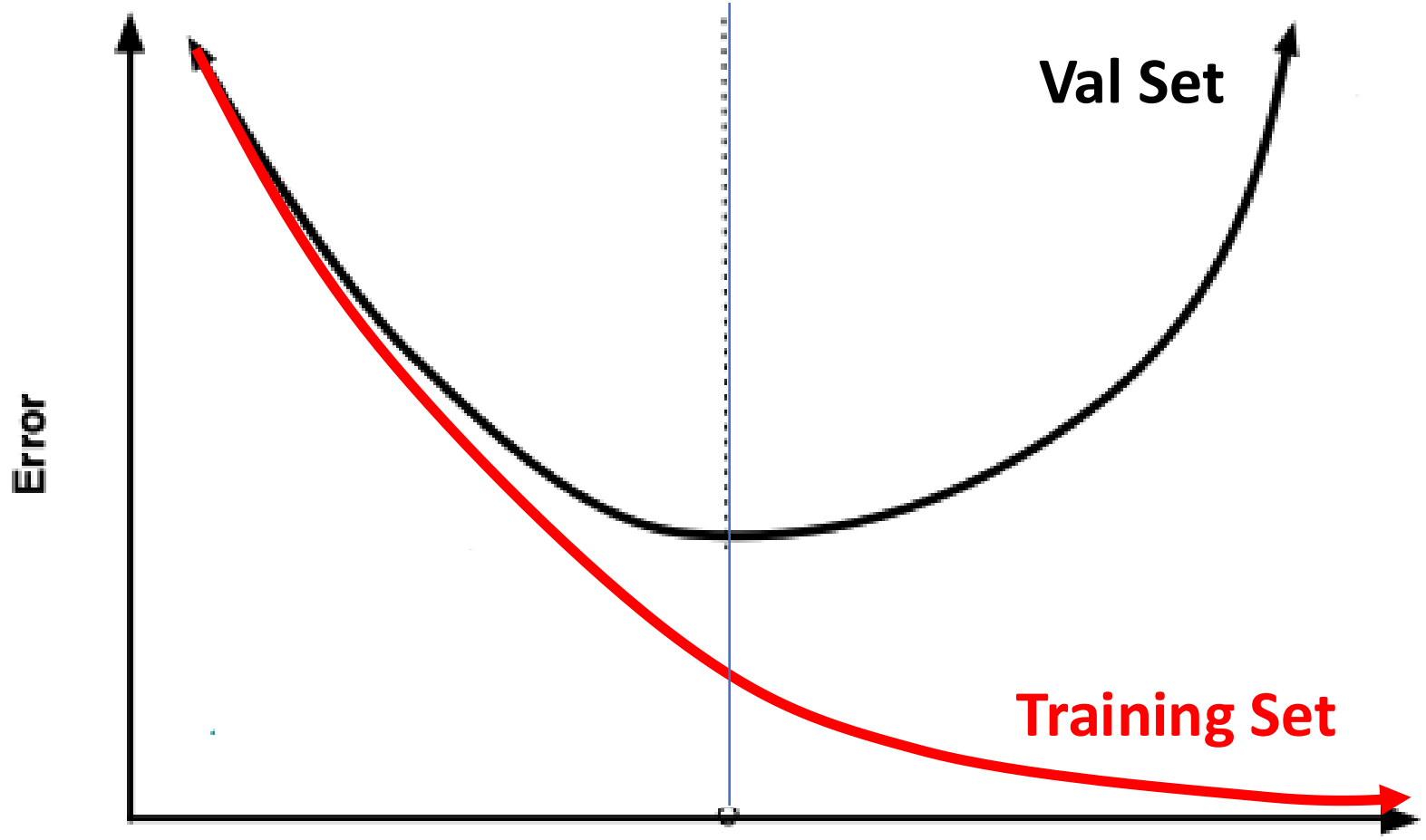
- 3. Weight-change criterion
  - Compare weights at epochs $(t-10)$ and t and test:

$$\max_i \left| w_i^t - w_i^{t-10} \right| < \theta$$

  - Don't base on length of overall weight change vector
  - Possibly express as a percentage of the weight
  - Be cautious: small weight changes at critical points can result in rapid drop in error

Slide Credit: Michael Mozer

Training Vs. Val Set Error



Slide Credit: Michael Mozer

Data Preprocessing and weight initialization

Data Preprocessing

- Mean subtraction
- Normalization
- PCA and whitening

Data Preprocessing: Mean subtraction

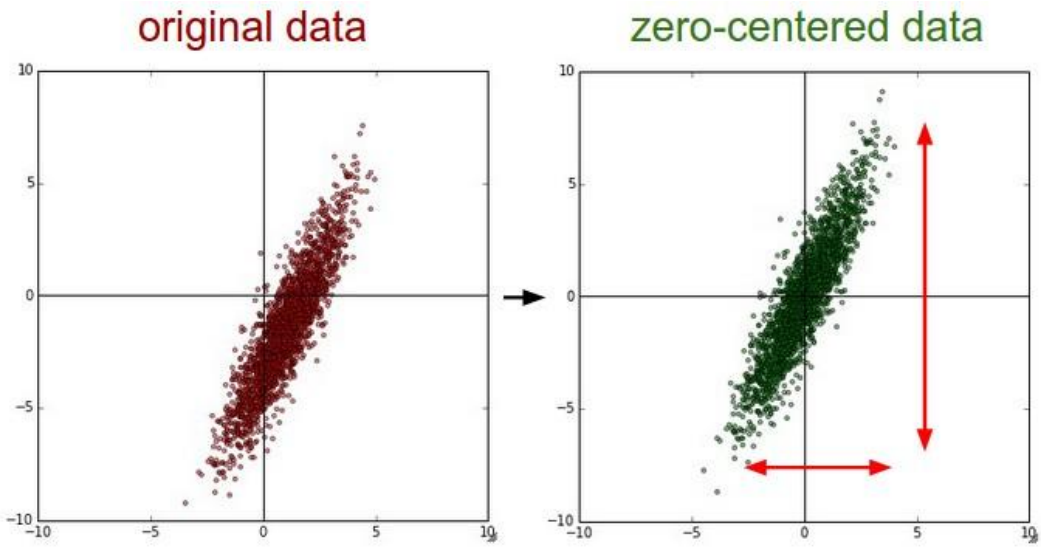
- Compute the mean of each dimension, μ_i, over the training set:

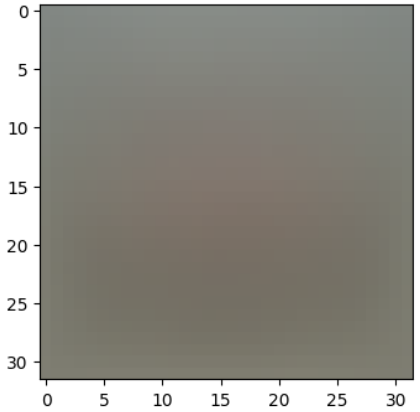
$$\mu_i = \frac{1}{N}\sum_j x_{ji}$$

- Subtract the mean for each dimension:

$$x'_{ji} \leftarrow x_{ji} - \mu_i$$

- Effect: Move the data center (mean) to coordinate center





Mean image of CIFAR10 (from PA1)

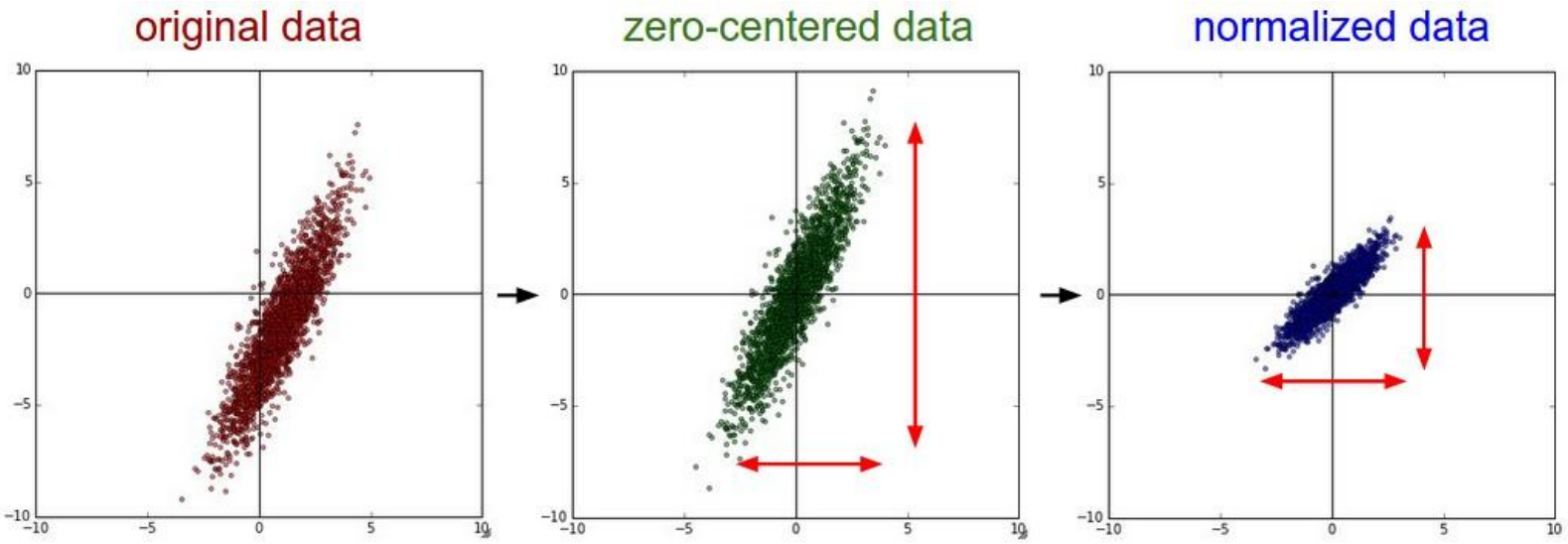
http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Normalization (or conditioning)

- Necessary if you believe that your dimensions have different scales
  - Might need to reduce this to give equal importance to each dimension
- Normalize each dimension by its std. dev. after mean subtraction:

$$x'_{ji} = x_{ji} - \mu_i$$
$$x''_{ji} = x'_{ji} / \sigma_i$$

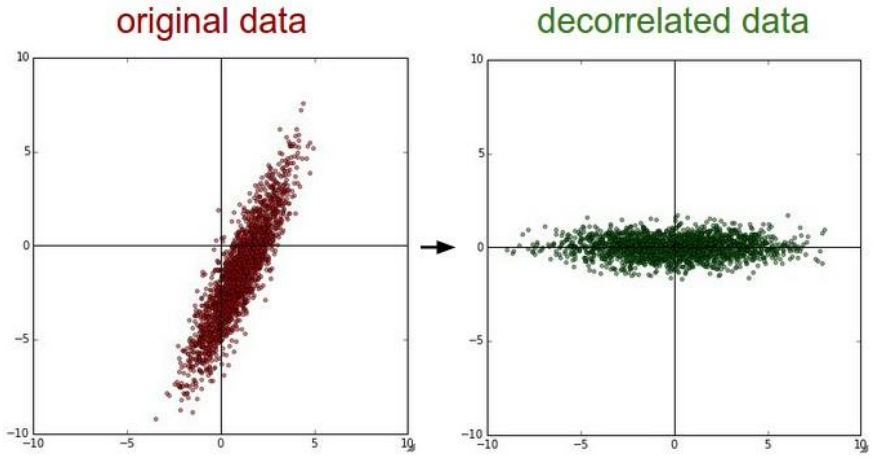
- Effect: Make the dimensions have the same scale



http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Principle Component Analysis

- First center the data
- Find the eigenvectors $e_1, \ldots, e_n$
- Project the data onto the eigenvectors:
  - $x_i^R = x_i \cdot [e_1, \ldots, e_n]$
- This corresponds to rotating the data to have the eigenvectors as the axes
- If you take the first M eigenvectors, it corresponds to dimensionality reduction



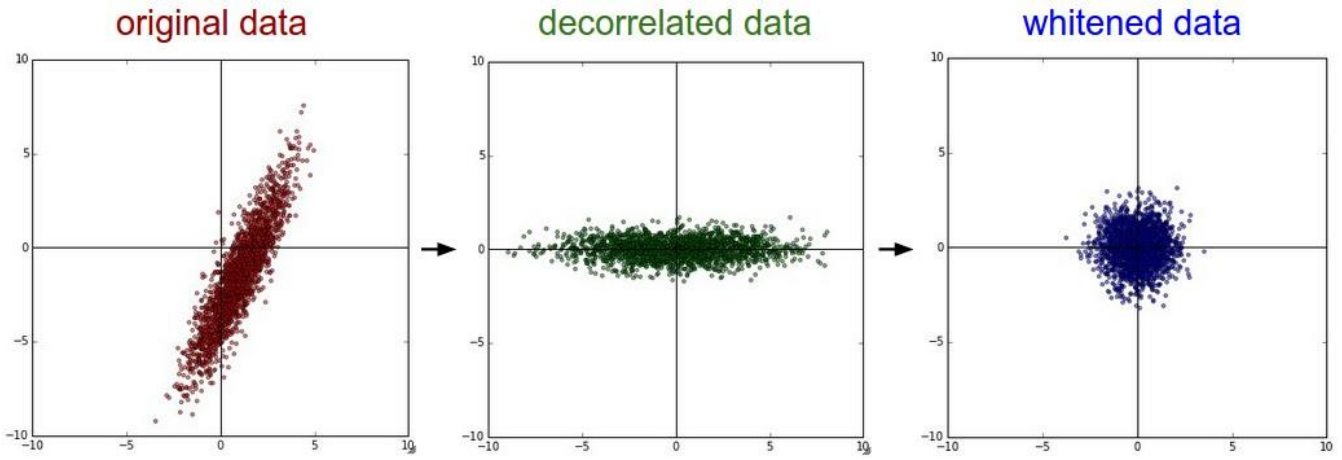
http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Whitening

- Normalize the scale with the norm of the eigenvalue:

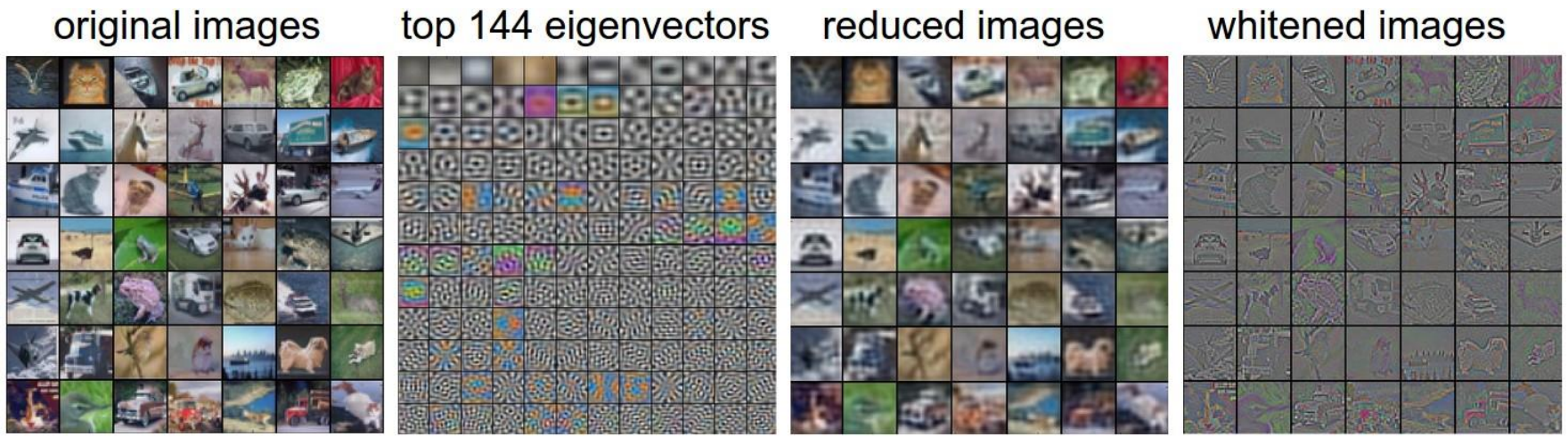
$$x_i^w = x_i^R / (\lambda_i + \epsilon)$$

- ϵ: a very small number to avoid division by zero
- This stretches each dimension to have the same scale.
- Side effect: this may exaggerate noise.



http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Example



http://cs231n.github.io/neural-networks-2/

Data Preprocessing: Summary

- We mostly don’t use PCA or whitening
  - They are computationally very expensive
  - Whitening has side effects
- It is quite crucial and common to zero-center the data
- Most of the time, we see normalization with the std. deviation

Weight Initialization

- Zero weights
  - Wrong!
  - Leads to updating weights by the same amounts for every input
  - Symmetry!
- Initialize the weights randomly to small values:
  - Sample from a small range, e.g., Normal(0,0.01)
  - Don’t initialize too small
- The bias may be initialized to zero
  - For ReLU units, this may be a small number like 0.01.

Note: None of these provide guarantees. Moreover, there is no guarantee that one of these will always be better.

Initial Weight Normalization

- Problem: Variance of the output changes with the number of inputs
- If $s = \sum_i w_i x_i$ (note that $Var(X) = E[(X - \mu)^2]$):

$$
\begin{aligned}
\mathrm{Var}(s) &= \mathrm{Var}\left(\sum_i^n w_i x_i\right) \\
&= \sum_i^n \mathrm{Var}(w_i x_i) \\
&= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + E[(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i) \\
&= (n\mathrm{Var}(w))\,\mathrm{Var}(x)
\end{aligned}
$$

Eqn: https://cs231n.github.io/neural-networks-2/#init

Initial Weight Normalization

- Solution:
  - Get rid of n in $Var(s) = (n$ Var(w))Var(x)

- How?
  - Scale the initial weights by $\sqrt{n}$
  - Why? Because: $Var(aX) = a^2 Var(X)$

- Standard Initialization (top plots in Figure 6 & 7):

$$w_i \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

which yields $n\,Var(w) = \frac{1}{3}$

because variance of $U[-r, r]$ is $\frac{r^2}{3}$ [1].

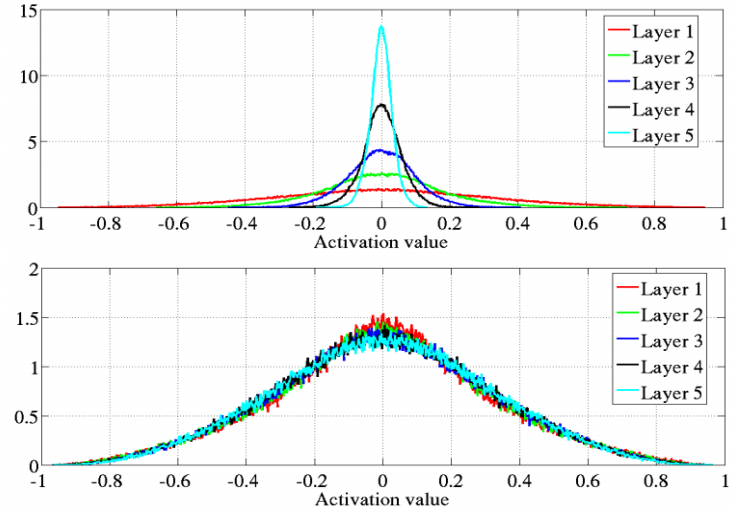


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

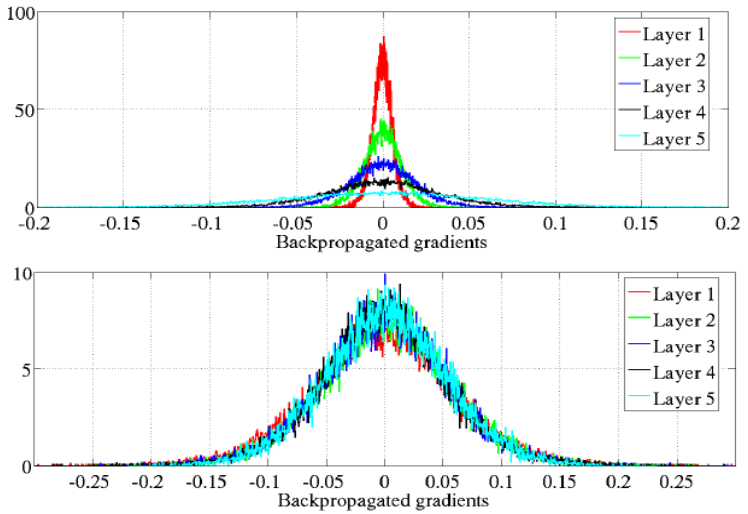


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Figures: Glorot & Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

Xavier initialization for symmetric activation functions (Glorot & Bengio):

$$w_i \sim N\left(0, \frac{\sqrt{2}}{\sqrt{n_{in} + n_{out}}}\right)$$

With Uniform distribution:

$$w_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right]$$

[1] https://proofwiki.org/wiki/Variance_of_Continuous_Uniform_Distribution

Initial Weight Normalization

He et al., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", 2015.

- He et al. shows that Xavier initialization does not work well for ReLUs.

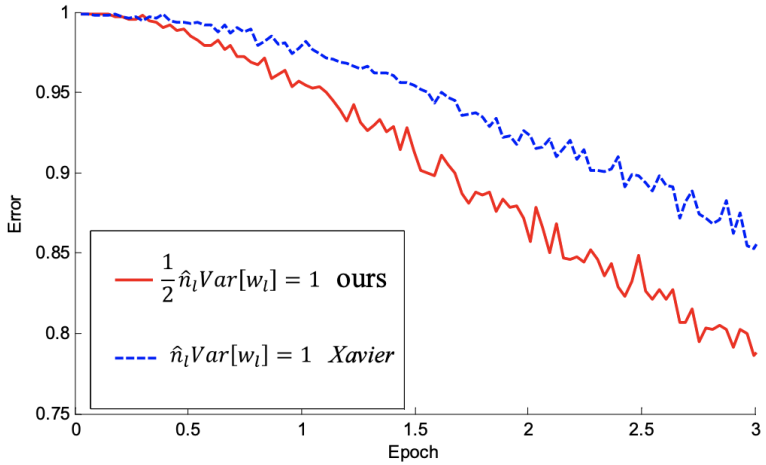


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and "*Xavier*" (blue) [7] lead to convergence, but ours starts reducing error earlier.

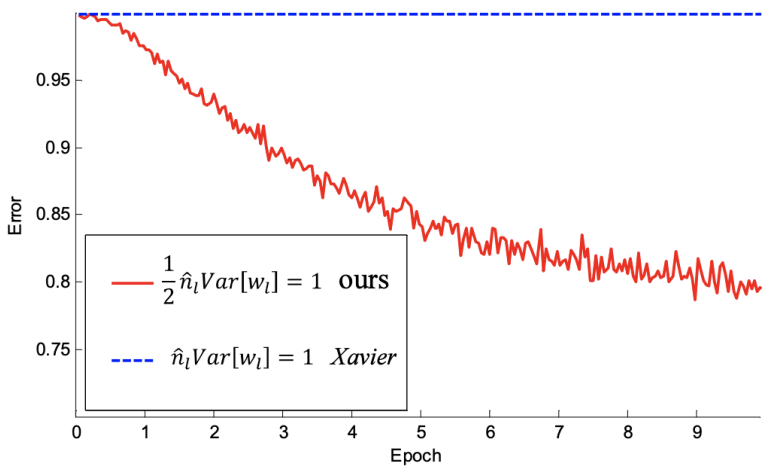


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But "*Xavier*" (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

More on Weight Initialization

Tutorial and Demo:

https://www.deeplearning.ai/ai-notes/initialization/index.html

Tutorial:

https://mmuratarat.github.io/2019-02-25/xavier-glorot-he-weight-init

Alternative: Batch Normalization

- Normalization is differentiable
  - So, make it part of the model (not only at the beginning)
  - I.e., perform normalization during every step of processing
- More robust to initialization
- Shown to also regularize the network in some cases (dropping the need for dropout)
- Issue: How to normalize at test time?
  1. Store means and variances during training, or
  2. Calculate mean & variance over your test data
- PyTorch: use model.eval() in test time.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Ioffe & Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", 2015.

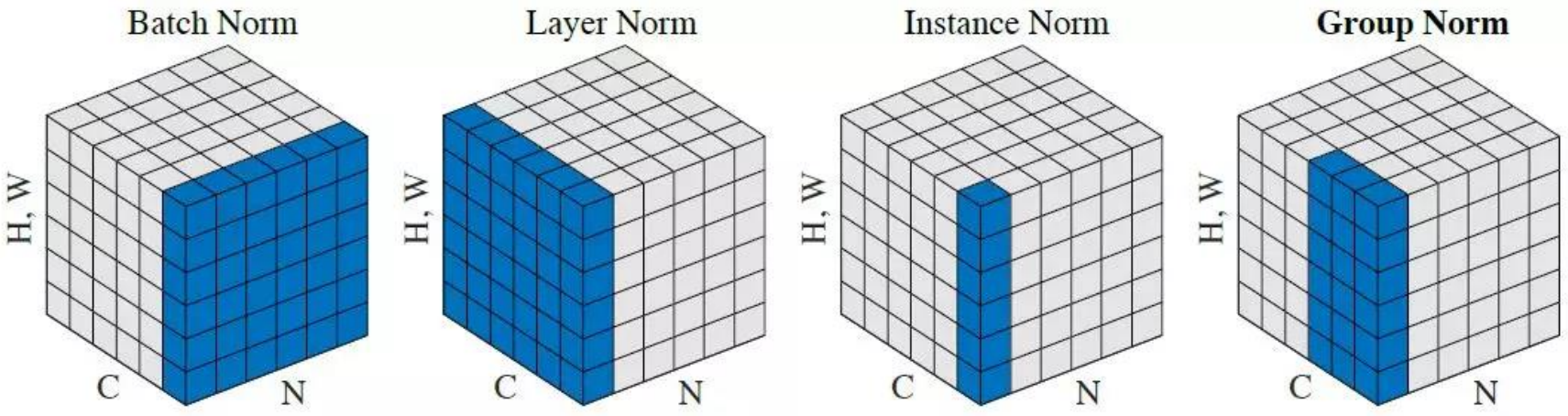
Alternative: Batch Normalization

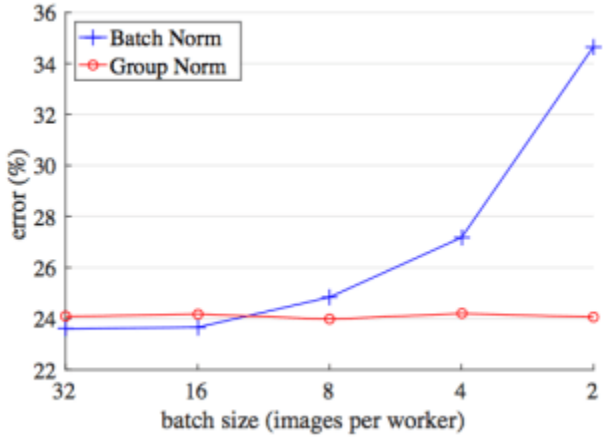
- Before or after non-linearity?
- Proposers
  - "If, however, we could ensure that the distribution of nonlinearity inputs remains more stable as the network trains, then the optimizer would be less likely to get stuck in the saturated regime, and the training would accelerate."
  - "As each layer observes the inputs produced by the layers below, it would be advantageous to achieve the same whitening of the inputs of each layer. By whitening the inputs to each layer, we would take a step towards achieving the fixed distributions of inputs that would remove the ill effects of the internal covariate shift."
  - "We add the BN transform immediately before the nonlinearity, by normalizing x = Wu+b. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, Wu + b is more likely to have a symmetric, non-sparse distribution, that is "more Gaussian" (Hyv¨arinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution."

BatchNorm introduces scale invariance

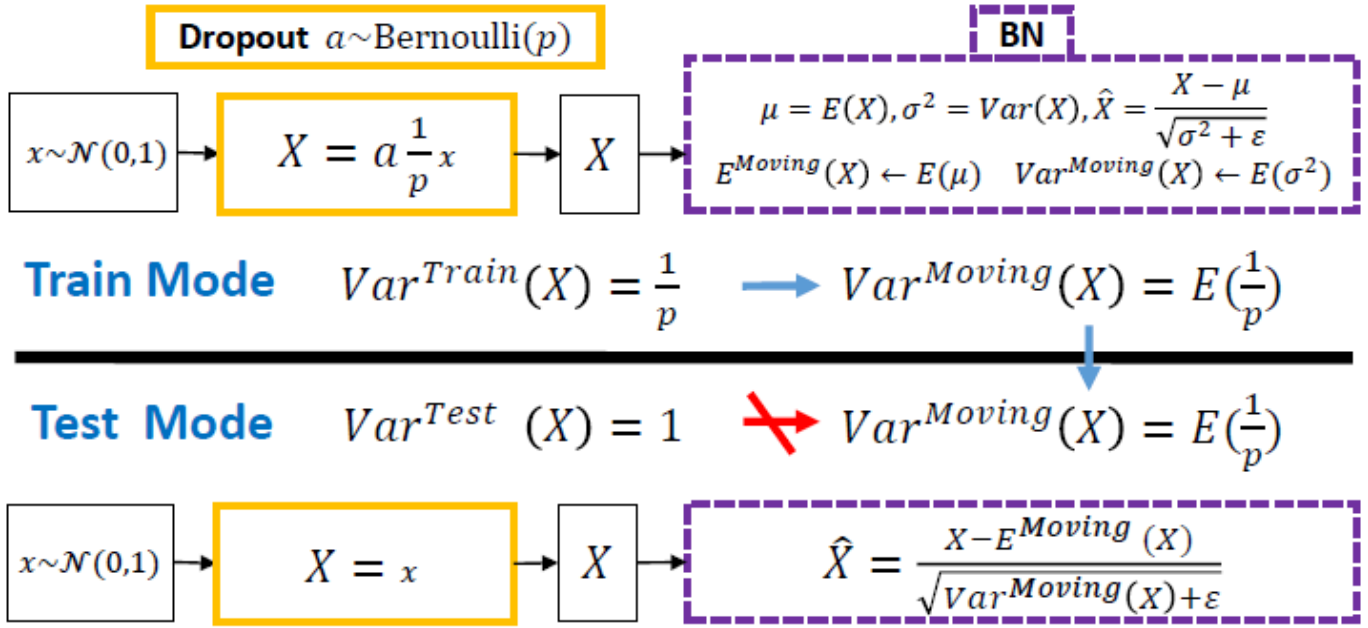
- https://www.inference.vc/exponentially-growing-learning-rate-implications-of-scale-invariance-induced-by-batchnorm/

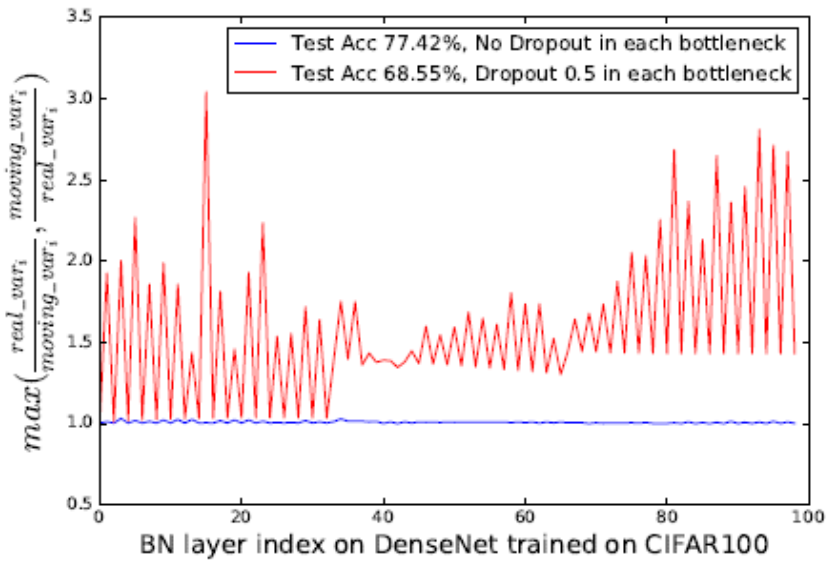
Alternative Normalizations





https://medium.com/syncedreview/facebook-ai-proposes-group-normalization-alternative-to-batch-normalization-fb0699bffae7





Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift

Xiang Li [1] Shuo Chen [1] Xiaolin Hu [2] Jian Yang [1]

2018

Since we get a clear knowledge about the disharmony between Dropout and BN, we can easily develop several approaches to combine them together, to see whether an extra improvement could be obtained. In this section, we introduce two possible solutions in modifying Dropout. One is to avoid the scaling on feature-map before every BN layer, by only applying Dropout after the last BN block. Another is to slightly modify the formula of Dropout and make it less sensitive to variance, which can alleviate the shift problem and stabilize the numerical behaviors.

How critical are BatchNorm parameters?

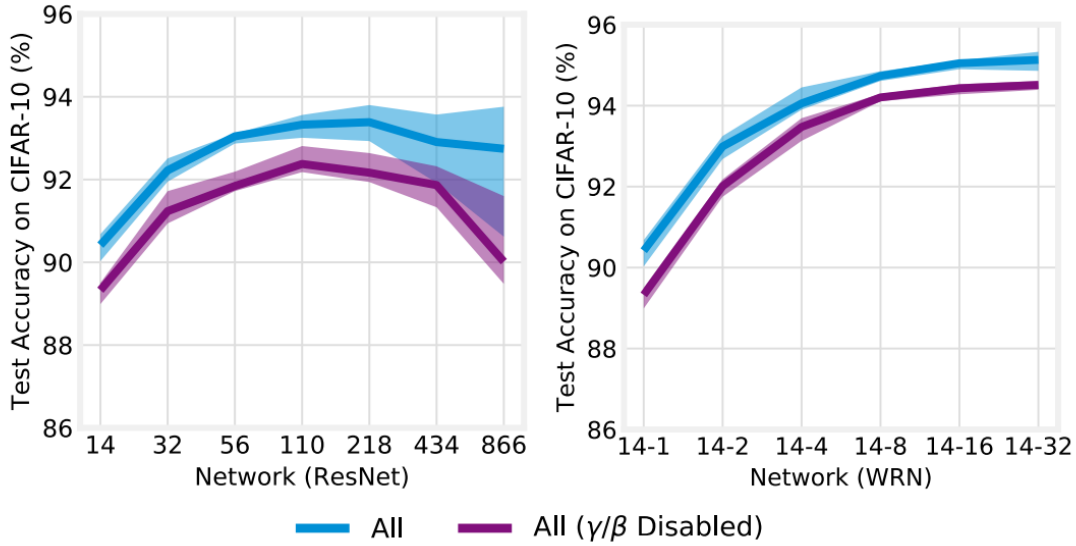


Figure 1. Test accuracy when training all parameters of the deep (left) and wide (right) ResNets in Table 1 with γ and β enabled (blue) and frozen at their initial values (purple). Except on the deepest ResNets, accuracy is about half a percent lower when γ and β are disabled.

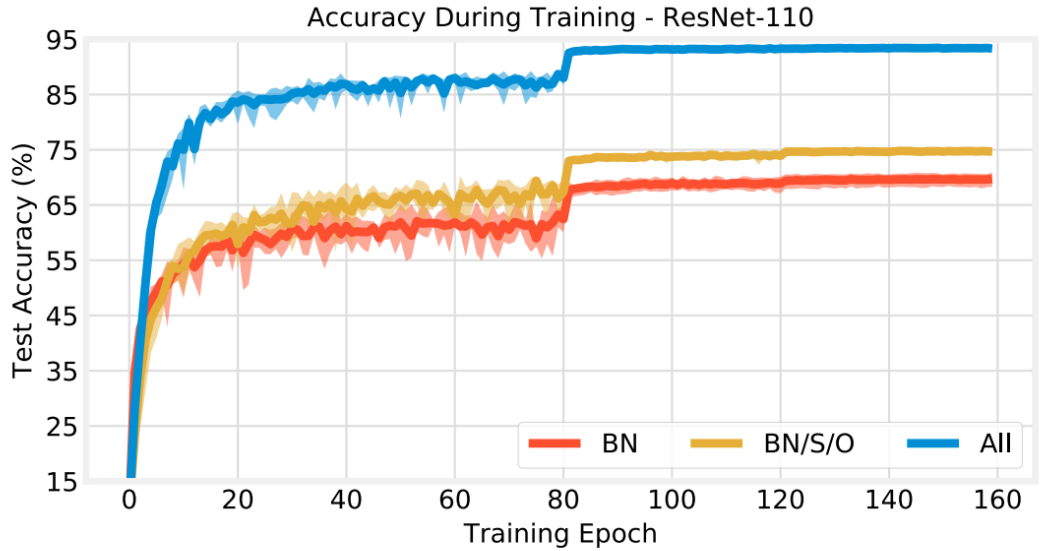


Figure 3. Test accuracy of ResNet-110 during training when training all parameters, just BatchNorm, and BatchNorm with shortcut and output parameters. Learning appears to occur at a similar rate in all experiments, although they reach different accuracies.

Frankle et al., "Training BatchNorm and Only BatchNorm: On the Expressive Power of Random Features in CNNs", 2020.

To sum up

- Initialization and normalization are crucial
- Different initialization & normalization strategies may be needed for different deep learning methods
  - E.g., in CNNs, normalization might be performed only on convolution etc.

Issues & Practical advices

Issues & tricks

- Vanishing gradient
  - Saturated units block gradient propagation (why?)
  - A problem especially present in recurrent networks or networks with a lot of layers
- Overfitting
  - Drop-out, regularization and other tricks.
- Tricks:
  - Unsupervised pretraining
- Batch normalization (each unit's preactivation is normalized)
  - Helps keeping the preactivation non-saturated
  - Do this for mini-batches (adds stochasticity)
  - Backprop needs to be updated

Unsupervised pretraining

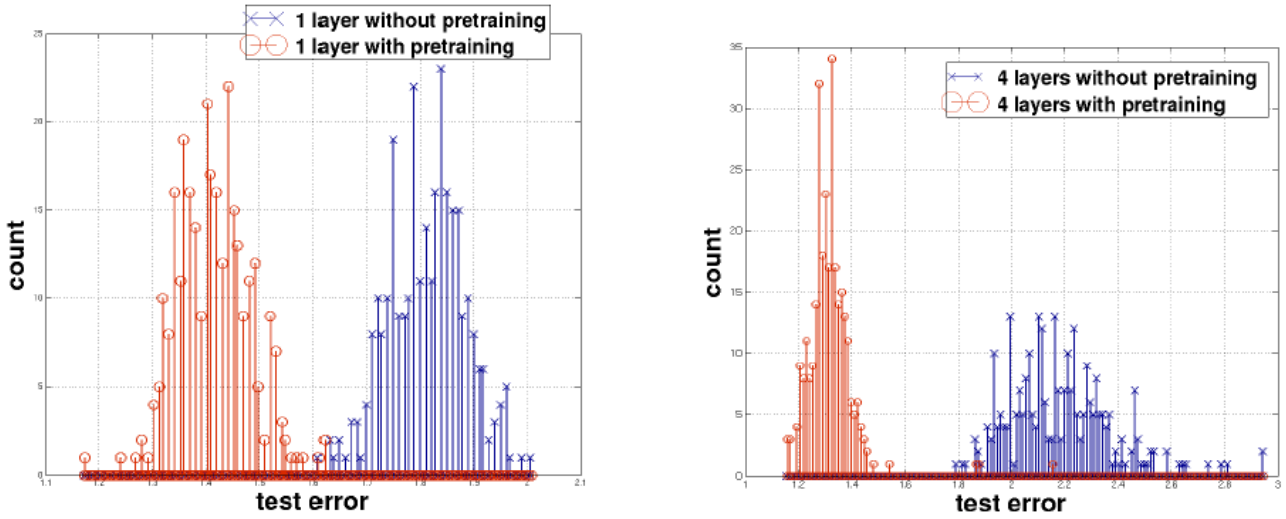


Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left**: 1 hidden layer. **Right**: 4 hidden layers.

Journal of Machine Learning Research 11 (2010) 625-660 Submitted 8/09; Published 2/10

Why Does Unsupervised Pre-training Help Deep Learning?

Dumitru Erhan* DUMITRU.ERHAN@UMONTREAL.CA
Yoshua Bengio YOSHUA.BENGIO@UMONTREAL.CA
Aaron Courville AARON.COURVILLE@UMONTREAL.CA
Pierre-Antoine Manzagol PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA
Pascal Vincent PASCAL.VINCENT@UMONTREAL.CA
Département d'informatique et de recherche opérationnelle
Université de Montréal
2920, chemin de la Tour
Montréal, Québec, H3T 1J8, Canada

Samy Bengio BENGIO@GOOGLE.COM
Google Research
1600 Amphitheatre Parkway
Mountain View, CA, 94043, USA

Unsupervised pretraining

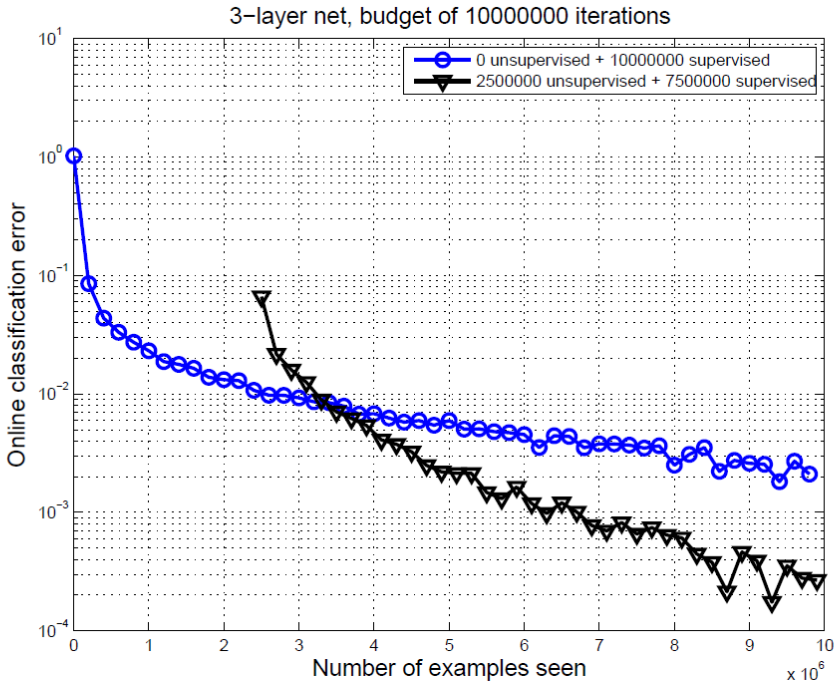


Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the beginning. The first 2.5 million examples are used for unsupervised pre-training (of a stack of denoising auto-encoders). The oscillations near the end are because the error rate is too close to zero, making the sampling variations appear large on the log-scale. Whereas with a very large training set regularization effects should dissipate, one can see that without pre-training, training converges to a poorer apparent local minimum: unsupervised pre-training helps to find a better minimum of the online error. Experiments performed by Dumitru Erhan.

Learning Deep Architectures for AI

Yoshua Bengio

What if things are not working?

- Check your gradients by comparing them against numerical gradients
  - More on this at: http://cs231n.github.io/neural-networks-3/
  - Check whether you are using an appropriate floating point representation
    - Be aware of floating point precision/loss problems
  - Turn off drop-out and other “extra” mechanisms during gradient check
  - This can be performed only on a few dimensions

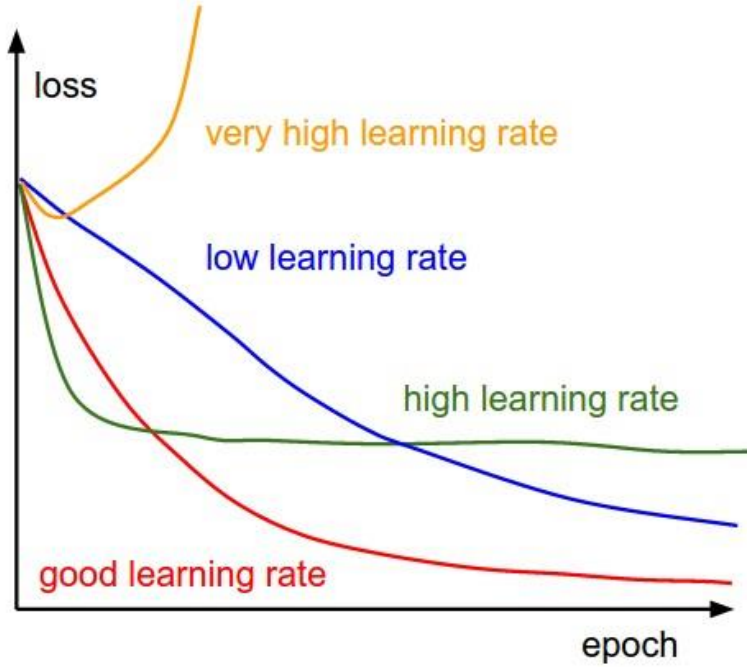
- Regularization loss may dominate the data loss
  - First disable regularization loss & make sure data loss works
  - Then add regularization loss with a big factor
  - And check the gradient in each case

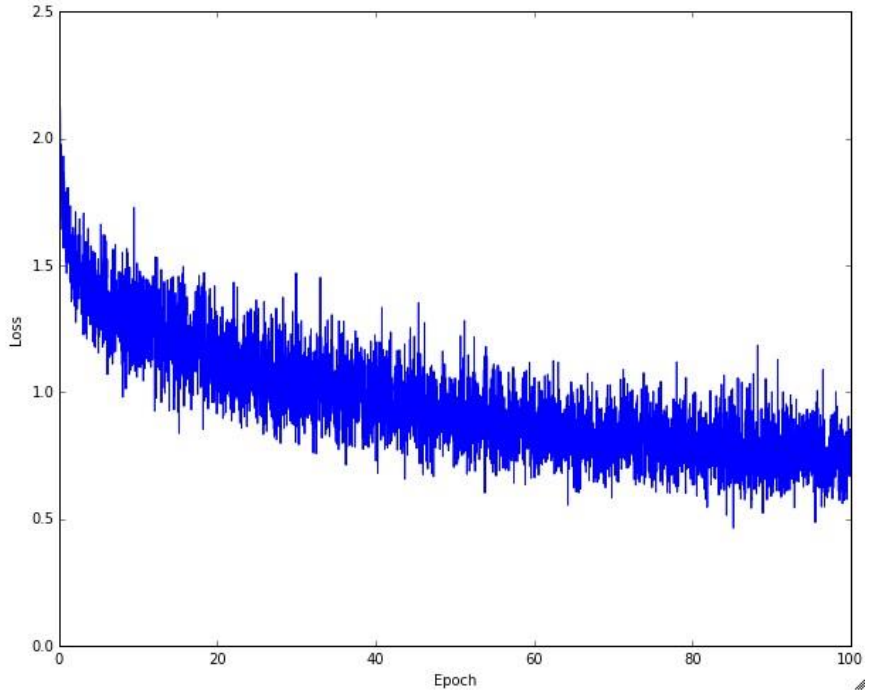
What if things are not working?

- Have a feeling of the initial loss value
  - For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is $-\ln(0.1)=2.302$
  - For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).

- Try to overfit on a tiny subset of the dataset
  - The cost should reach to zero if things are working properly

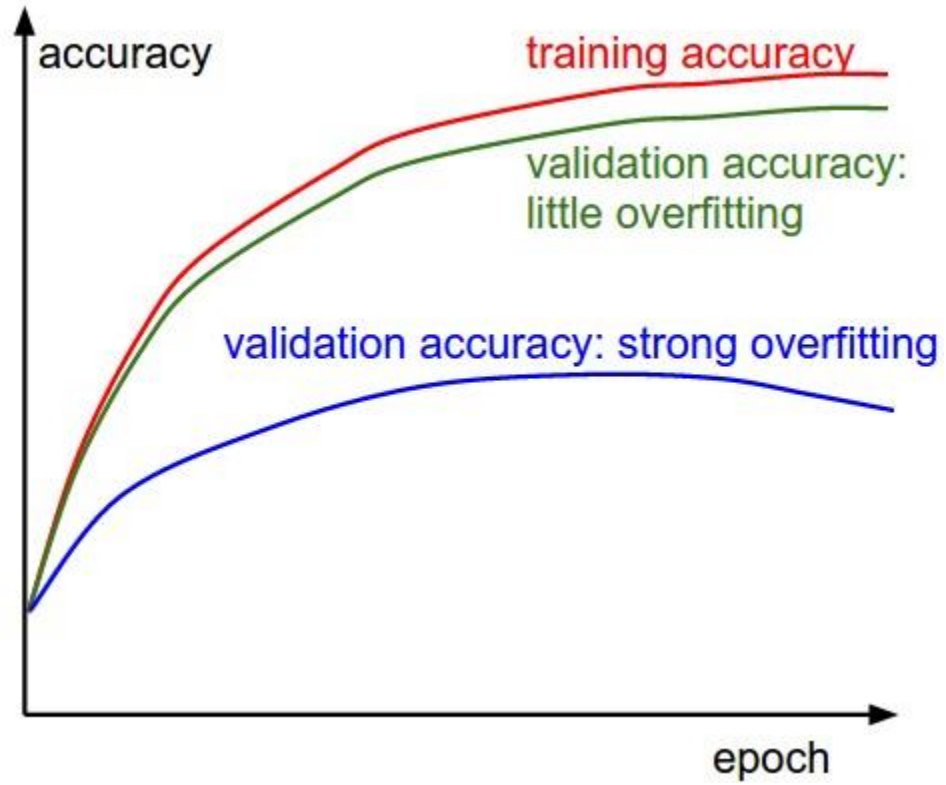
What if things are not working?





Learning rate might be too low;
Batch size might be too small

What if things are not working?

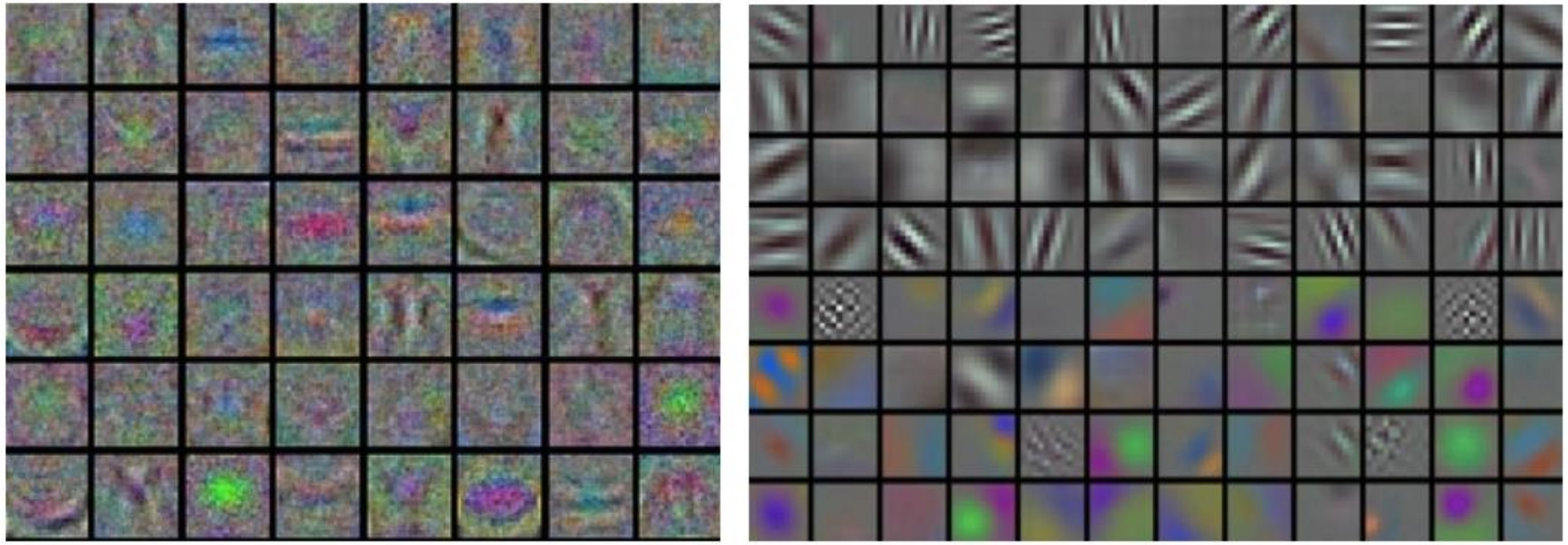


What if things are not working?

- Plot the histogram of activations per layer
  - E.g., for tanh functions, we expect to see a diverse distribution of values between [-1,1]

What if things are not working?

- Visualize your layers (the weights)



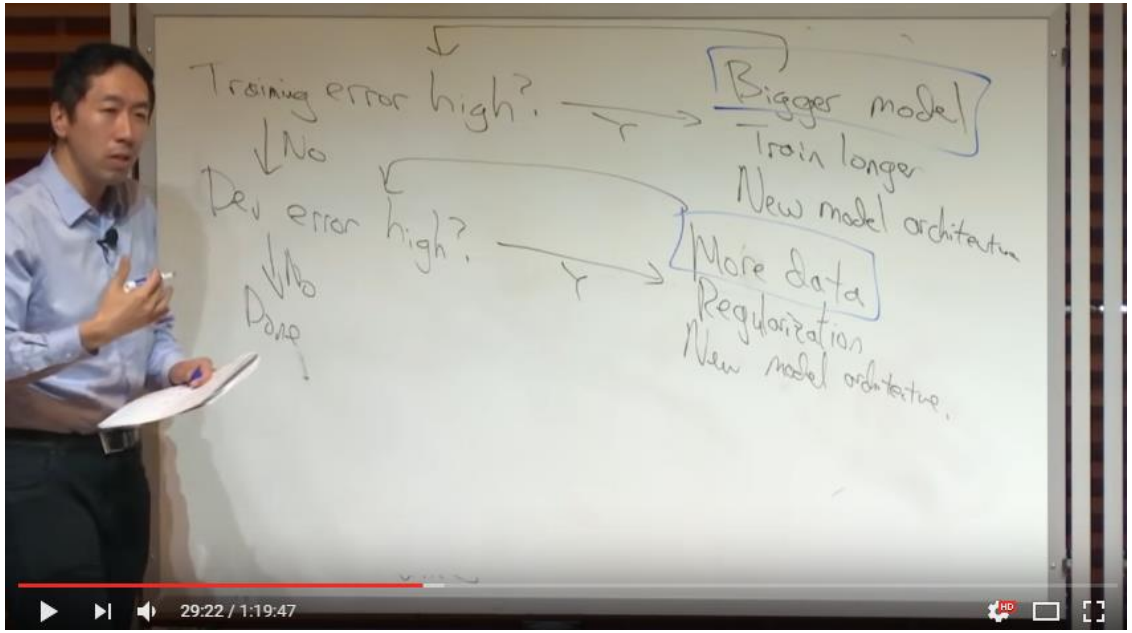
Examples of visualized weights for the first layer of a neural network. **Left**: Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right**: Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

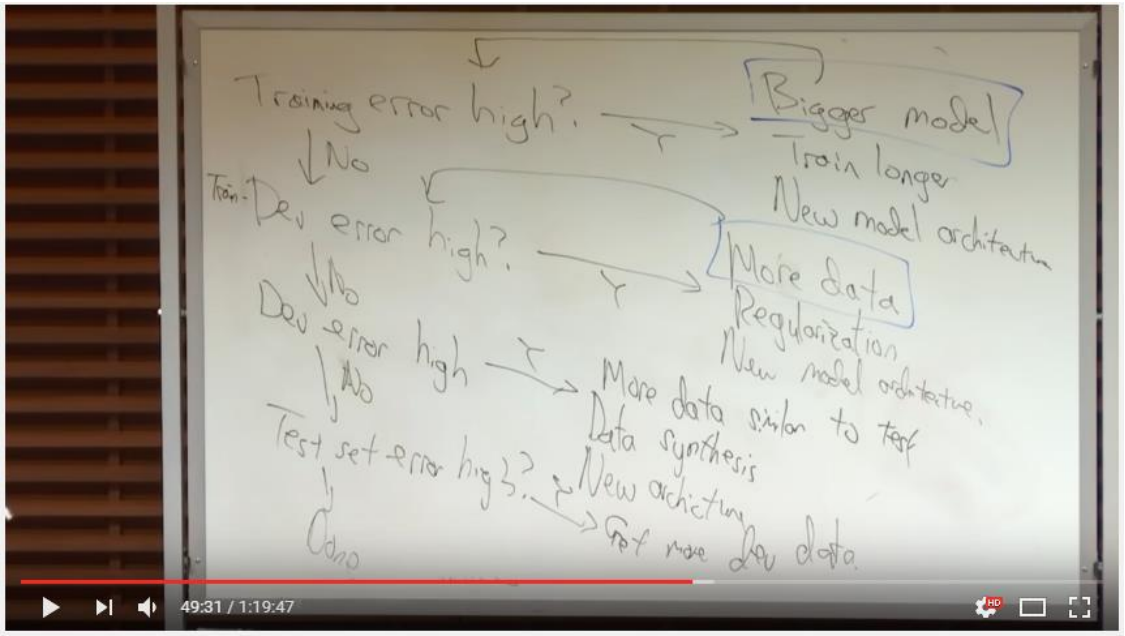
Andrew Ng's suggestions

https://www.youtube.com/watch?v=F1ka6a13S9I

- "In DL, the coupling between bias & variance is weaker compared to other ML methods:
  - We can train a network to have high bias and variance."

- "Dev (validation) and test sets should come from the same distribution. Dev&test sets are like problem specifications.
  - This requires especially attention if you have a lot of data from simulated environments etc. but little data from the real test environment."



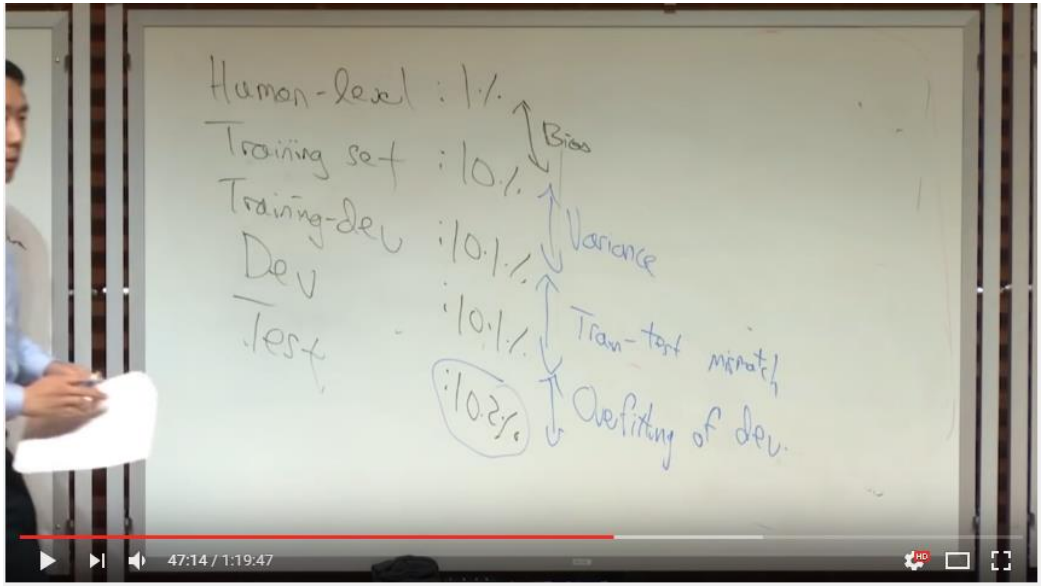


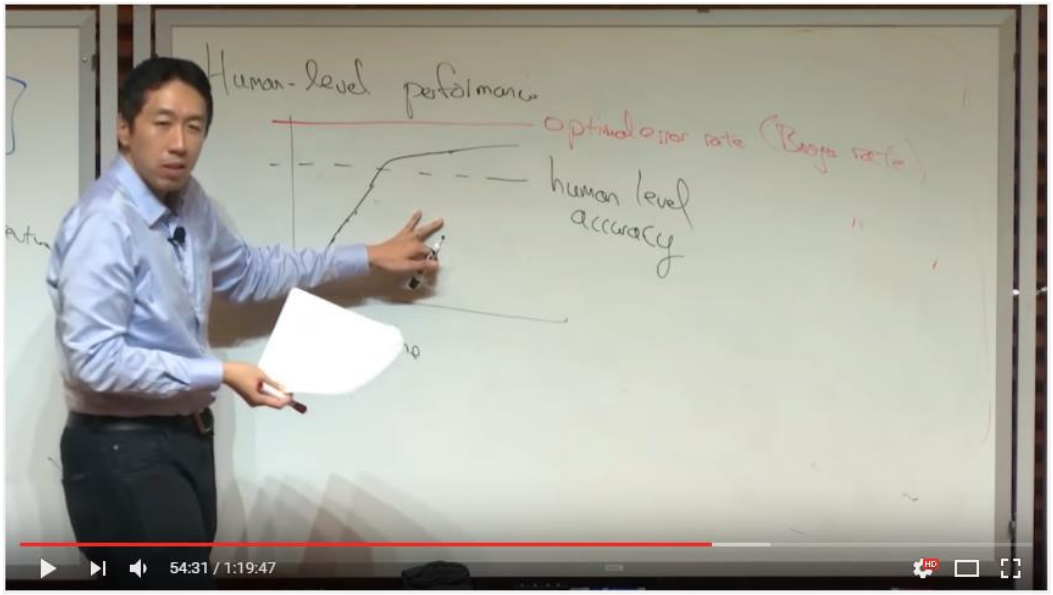
Andrew Ng's suggestions

https://www.youtube.com/watch?v=F1ka6a13S9I

- "Knowing the human performance level gives information about the problem of your network:
  - If training error is far from human performance, then there is a bias error.
  - If they are close but validation has more error (compared to the diff between human and training error), then there is variance problem."

- "After surpassing human level, performance increases only very slowly/difficult.
  - One reason: There is not much space for improvement (only tiny little details). Problem gets much harder.
  - Another reason: We get labels from humans."





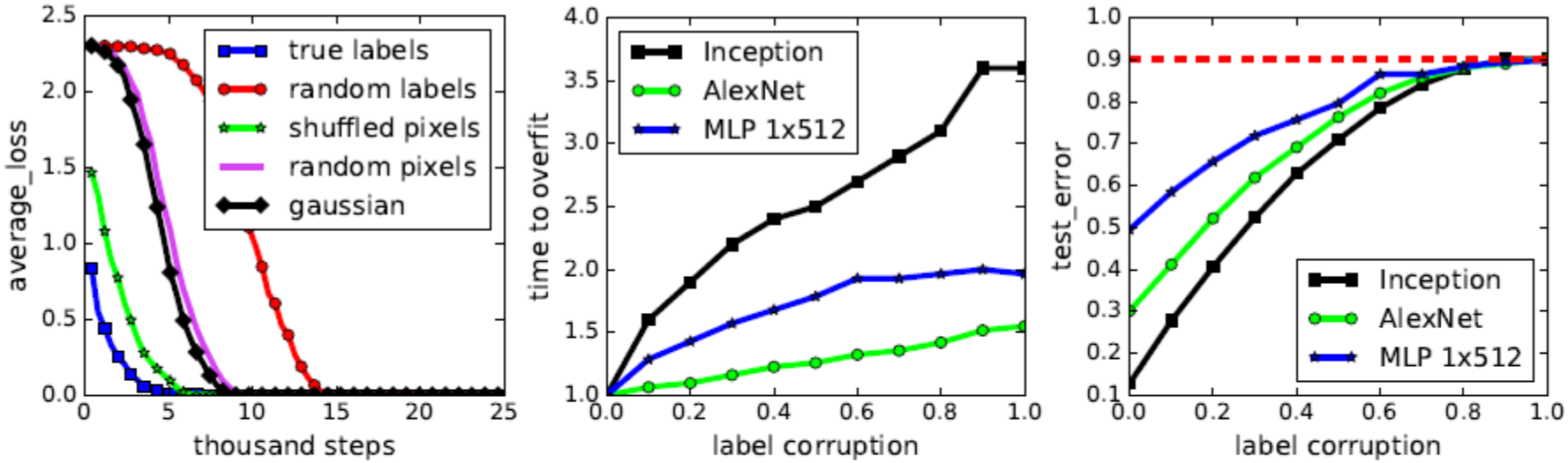
Also read the following

- 37 reasons why your neural network is not working:
  - https://medium.com/@slavivanov/4020854bd607
- “A Recipe for Training Neural Networks” by Karpathy:
  - http://karpathy.github.io/2019/04/25/recipe/
- Deep Learning Tuning Playbook:
  - https://github.com/google-research/tuning_playbook
- Calibrated Chaos: Variance Between Runs of Neural Network Training is Harmless and Inevitable:
  - https://arxiv.org/pdf/2304.01910.pdf

What is best then?

- Which algorithm to choose?
  - No answer yet
  - See Tom Schaul (2014)
  - Adam, RMSprop seem to be slightly favorable; however, no best algorithm
- SGD, SGD+momentum, RMSprop, RMSprop+momentum, Adam are the most widely used ones

Luckily, deep networks are very powerful



(a) learning curves (b) convergence slowdown (c) generalization error growth

Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training error is 0) under different label corruptions.

Regularization is turned off in the experiments. When you turn on regularization, the networks perform worse.

UNDERSTANDING DEEP LEARNING REQUIRES RE-THINKING GENERALIZATION

Chiyuan Zhang*
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

Benjamin Recht†
University of California, Berkeley
recht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

Concluding remarks for this part

- Loss functions
- Gradients of loss functions for minimizing them
  - All operations in the network should be differentiable
- Gradient descent and its variants
- Initialization, normalization, adaptive learning rate, …
- Overall, you have learned most of the tools you will use in the rest of the course.

Convolutional Neural networks: MOTIVATION

Disadvantages of MLPs: Dimensionality

- The number of parameters in an MLP is high for practical problems
  - e.g., for grayscale images with 1000x1000 resolution, a fully-connected layer with 1000 neurons requires 10^9 parameters.
- The number of parameters in an MLP increases quadratically with an increase in input dimensionality
- For example, for a fully-connected layer with n_{in} input neurons and n_{out} output neurons:
  - Number of parameters: $n_{in} \times n_{out}$
  - Assuming proportional decrease in layer size, e.g. $n_{out} = n_{in}/10$, gives: $n_{in} \times n_{out} = n_{in}^2/10$
  - Increasing n_{in} by d yields a change of $\mathcal{O}(d^2)$.
- This is a problem because:
  - More parameters => larger model size & more computational complexity.
- Teaser for CNNs:
  - Input size does not affect model size (in general)

Disadvantages of MLPs: Curse of Dimensionality

- For conventional ML methods:
  - The number of required samples for obtaining small error increases exponentially with input dimensions

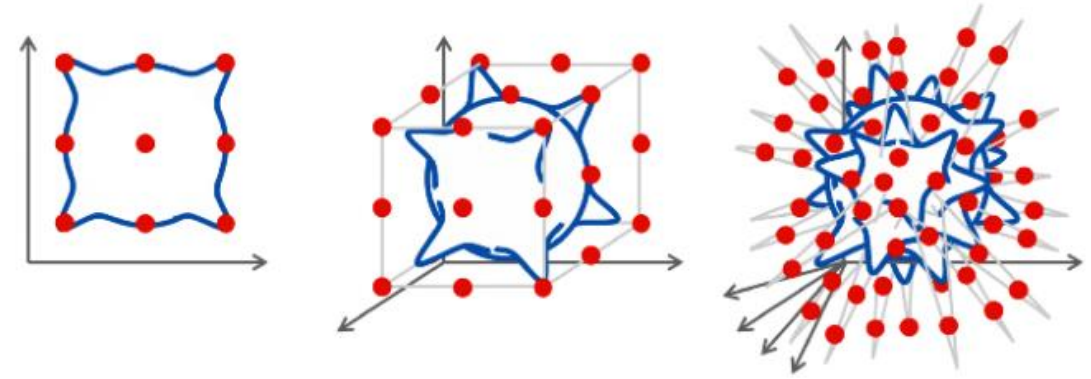


Illustration of the curse of dimensionality: in order to approximate a Lipschitz-continuous function composed of Gaussian kernels placed in the quadrants of a d-dimensional unit hypercube (blue) with error ε, one requires $\mathcal{O}(1/\varepsilon^d)$ samples (red points).

Figure: https://towardsdatascience.com/geometric-foundations-of-deep-learning-94cdd45b451d

- For deep networks:
  - This does not seem to be an issue for deep networks (see e.g. Poggio & Liao, 2018).

Poggio, T., & Liao, Q. (2018). Theory I: Deep networks and the curse of dimensionality. Bulletin of the Polish Academy of Sciences: Technical Sciences, (6).

Disadvantages of MLPs: Equivariance

- Vectorizing an image breaks patterns in consecutive pixels.
  - Shifting one pixel means a whole new vector
  - Makes learning more difficult
  - Requires more data to generalize

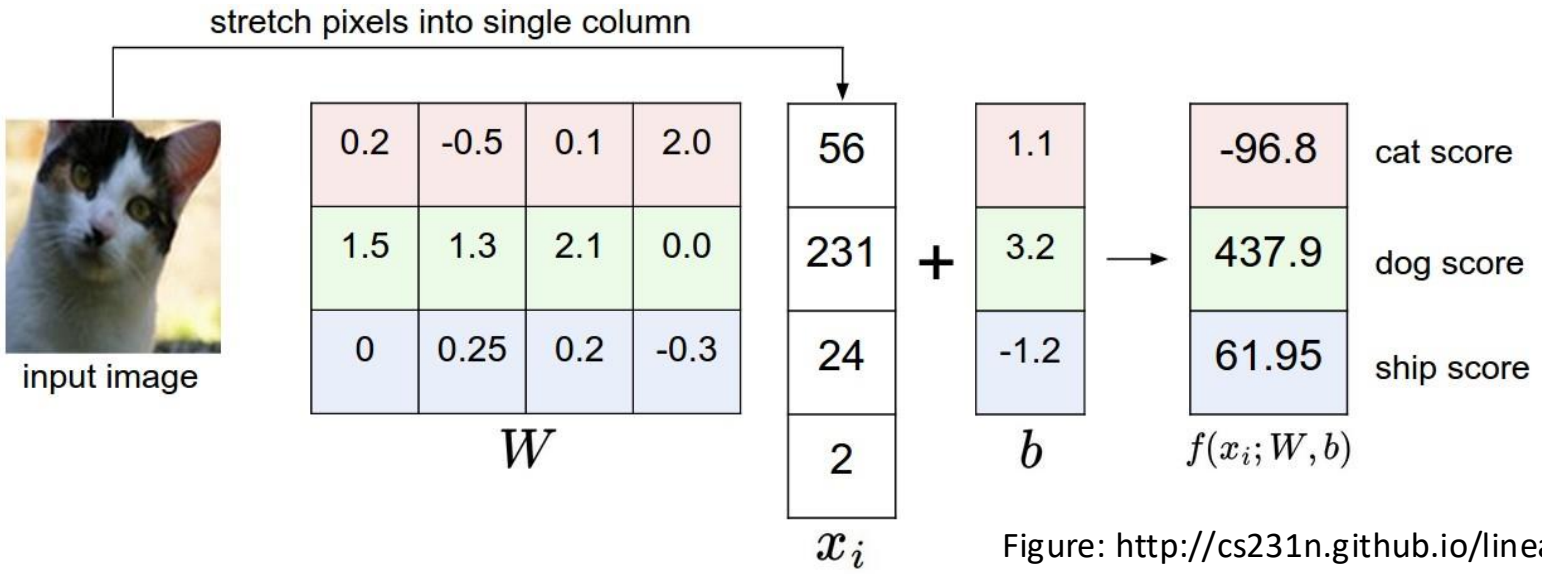
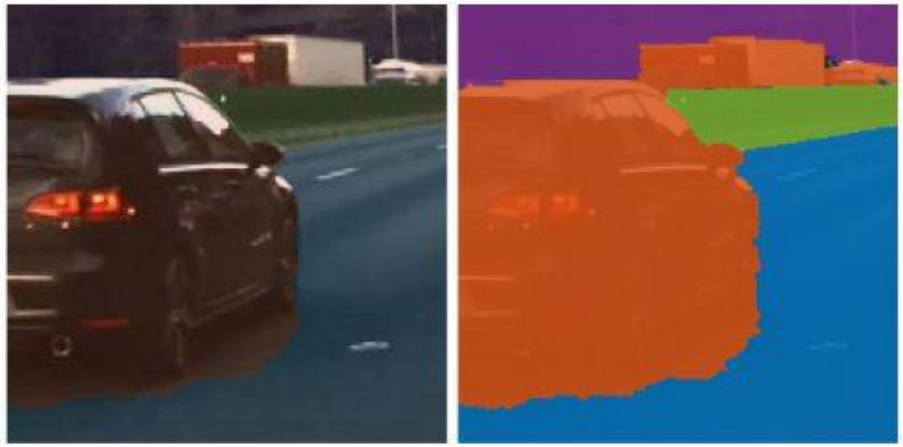


Figure: http://cs231n.github.io/linear-classify/

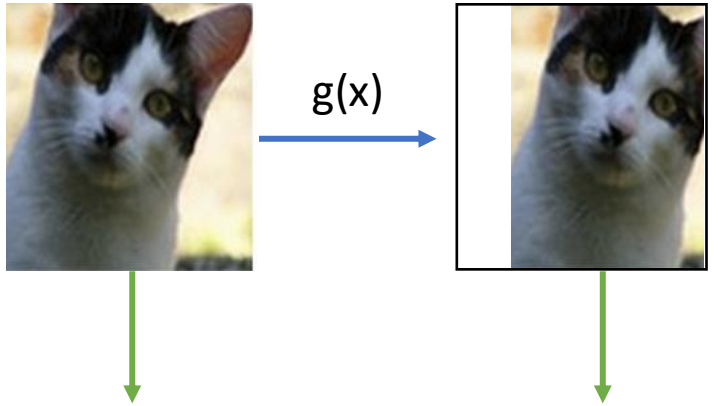
Equivariance vs. Invariance

- Equivariant problem: image segmentation.
  - f(g(x)) = g(f(x))



https://www.mathworks.com/discovery/image-segmentation.html

- Invariant problem: object recognition.
  - f(g(x)) = f(x)

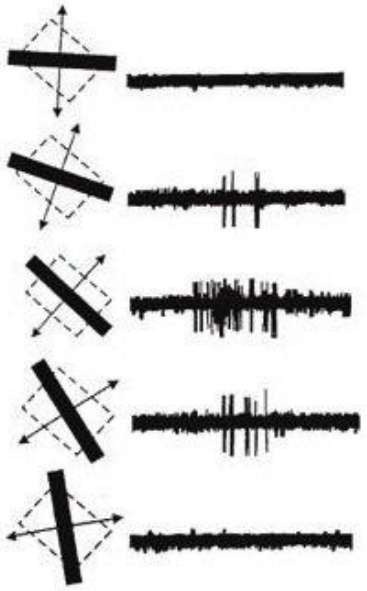


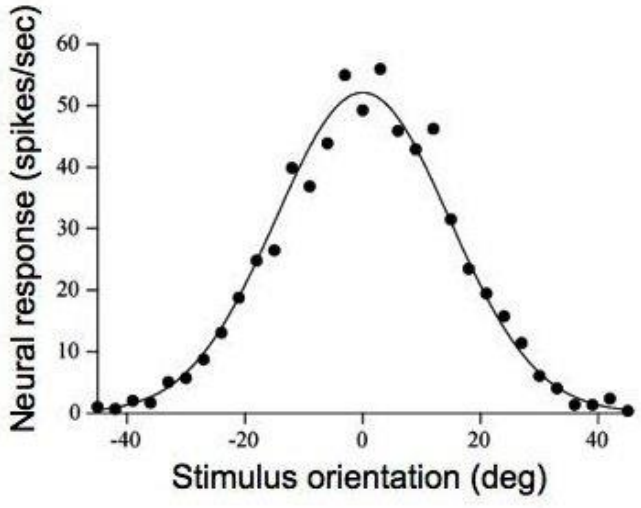
- Pooling provides invariance, convolution provides equivariance.

An Alternative to MLPs

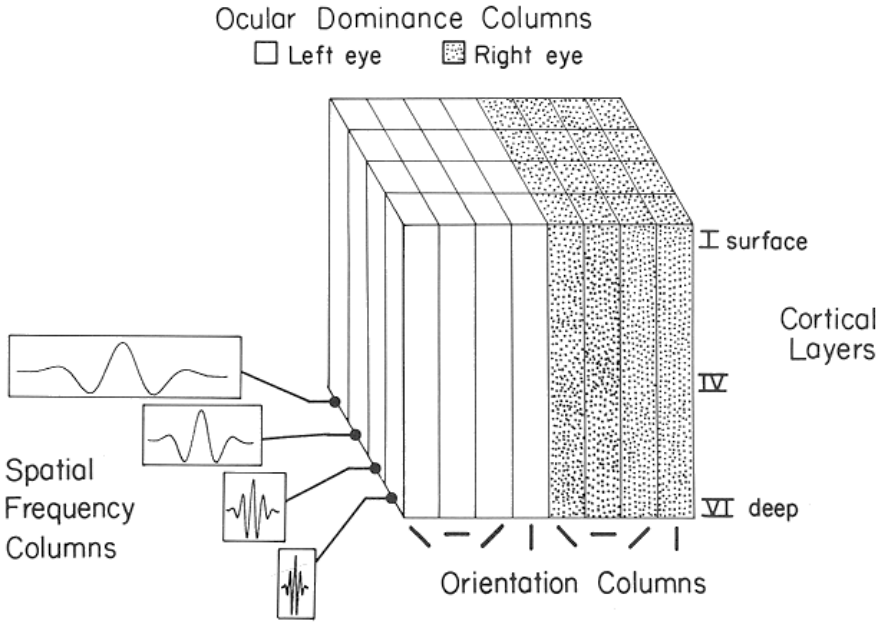
Solution (inspiration):

- Hubel & Wiesel: Brain neurons are not fully connected. They have local receptive fields





Hubel & Wiesel, 1968



Model of Striate Module in Cats

http://fourier.eng.hmc.edu/e180/lectures/retina/node1.html

An Alternative to MLPs

Solution (inspiration):

- Hubel & Wiesel: Brain neurons are not fully connected. They have local receptive fields

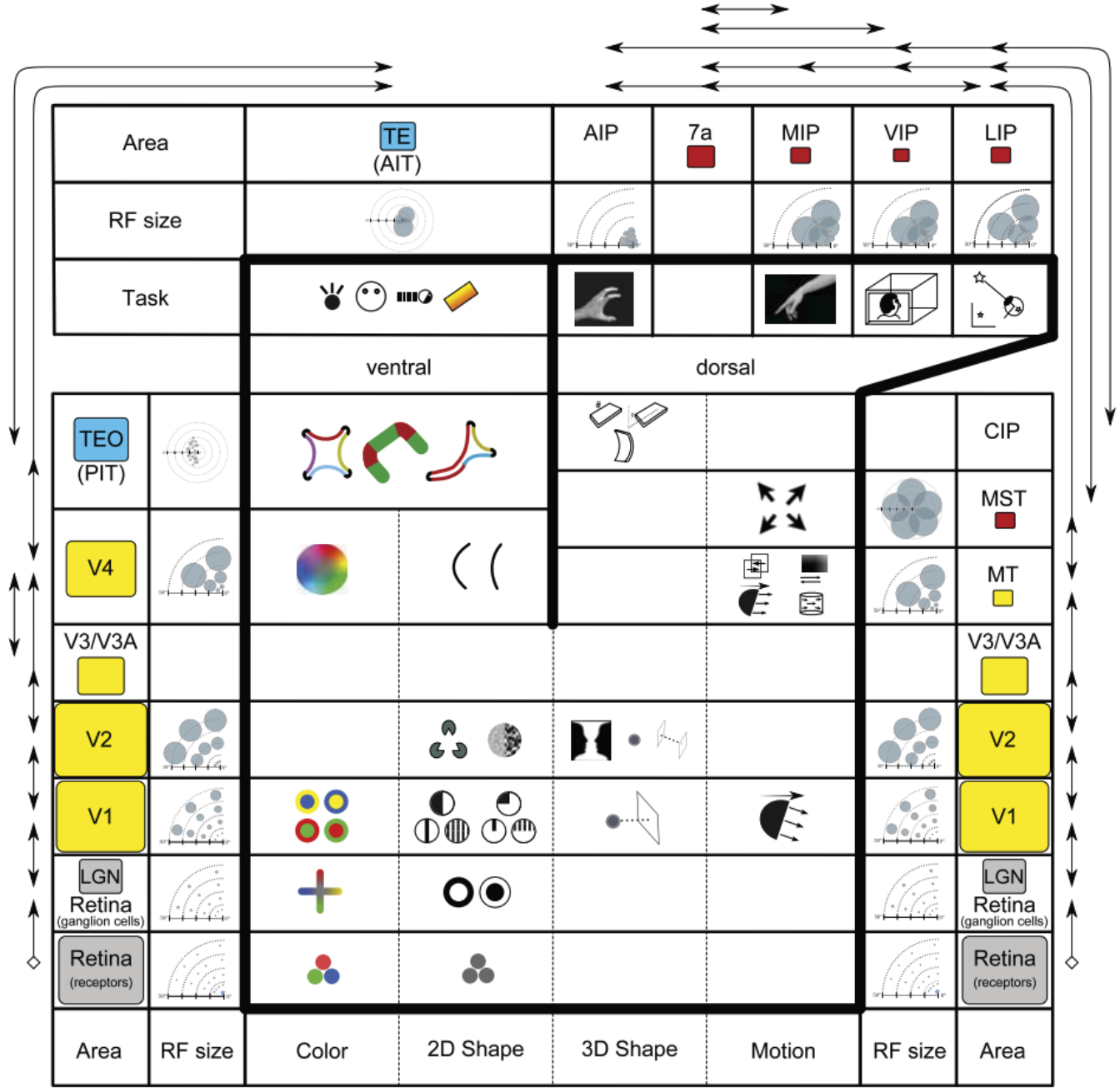


Figure: N. Krueger, P. Janssen, S. Kalkan, M. Lappe, A. Leonardis, J. Piater, A. J. Rodriguez-Sanchez, L. Wiskott, "Deep Hierarchies in the Primate Visual Cortex: What Can We Learn For Computer Vision?", IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 2013.

An Alternative to MLPs

Solution: Neocognitron (Fukushima, 1979):

A neural network model unaffected by shift in position, applied to Japanese handwritten character recognition.

- S (simple) cells: local feature extraction.
- C (complex) cells: provide tolerance to deformation, e.g. shift.
- Self-organized learning method.

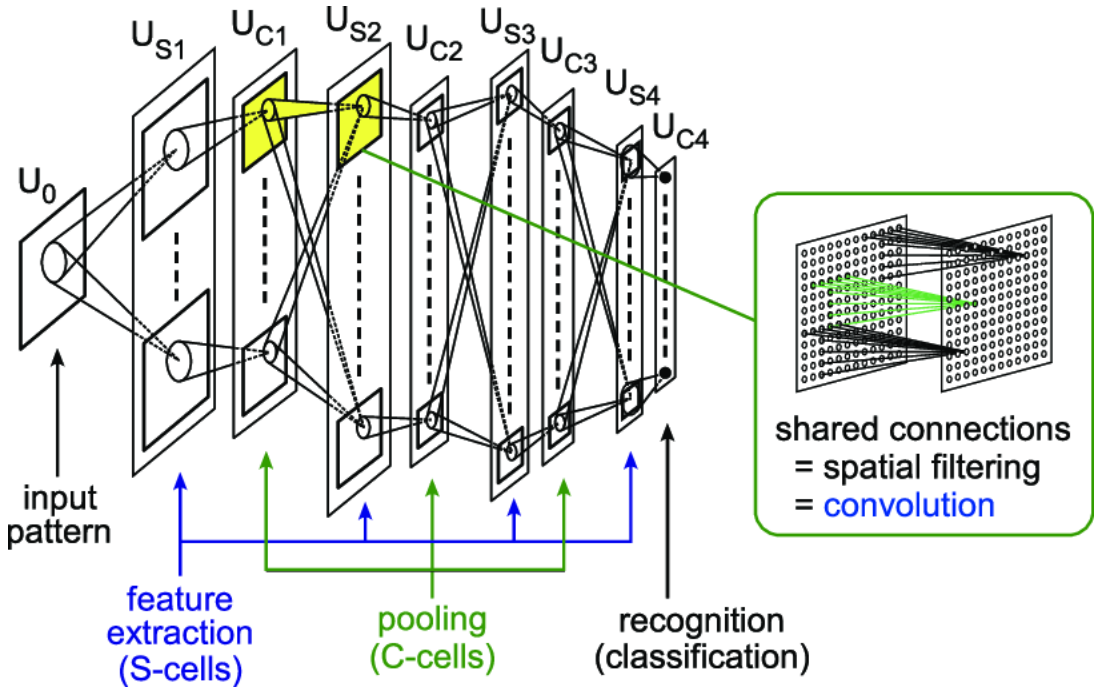


Figure: Fukushima (2019), Recent advances in the deep CNN neocognitron.

An Alternative to MLPs

Solution:

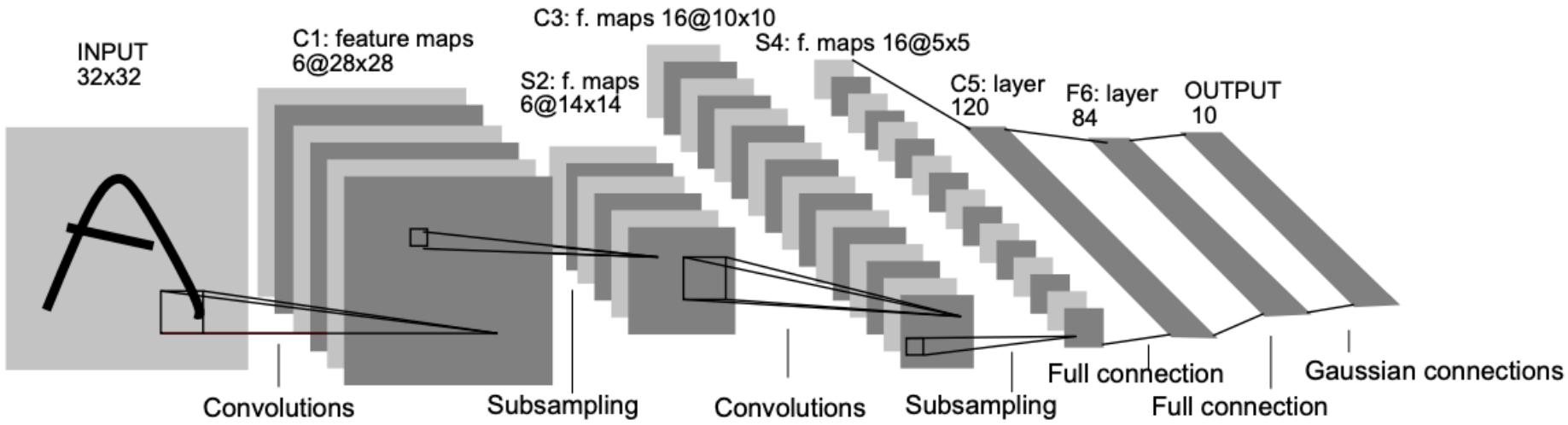
Neocognitron's self-organized learning method (Fukushima, 2019):

"For training intermediate layers of the neocognitron, the learning rule called AiS (Add-if-Silent) is used. Under the AiS rule, a new cell is generated and added to the network if all postsynaptic cells are silent in spite of non-silent presynaptic cells. The generated cell learns the activity of the presynaptic cells in one-shot. Once a cell is generated, its input connections do not change any more. Thus the training process is very simple and does not require time-consuming repetitive calculation."

An Alternative to MLPs

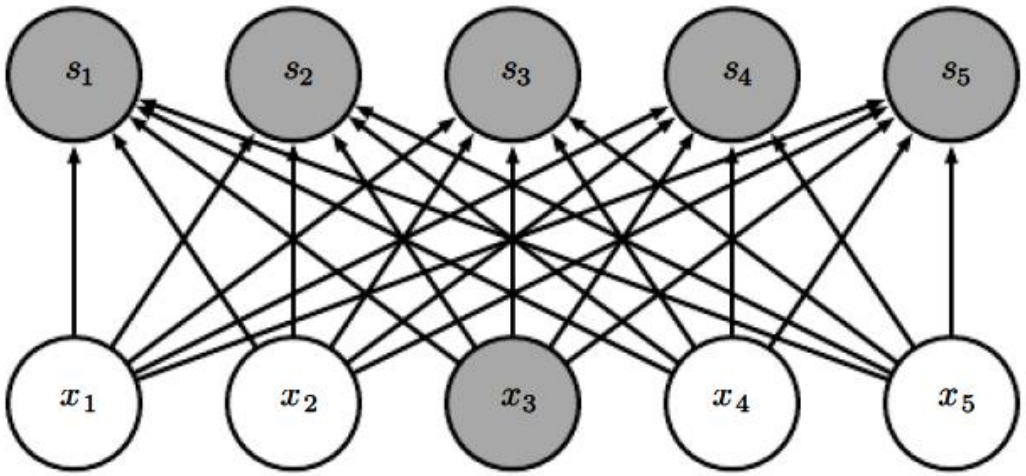
Solution: Convolutional Neural Networks (Lecun, 1998)

- Gradient descent
- Weights shared
- Document recognition



Lecun, 1998

CNNs: Underlying Principle



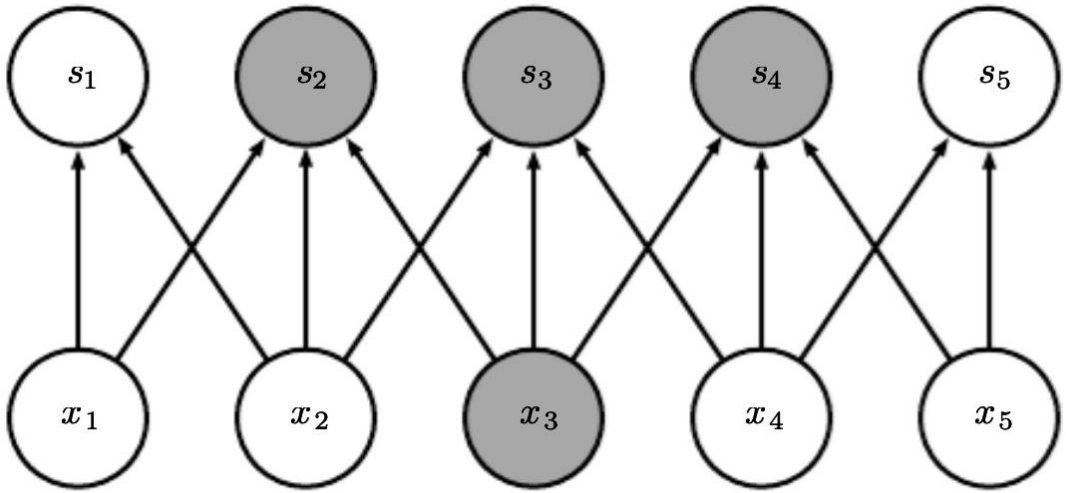


Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

CNNs vs. MLPs: Curse of Dimensionality

- A fully-connected network has too many parameters
  - On CIFAR-10:
    - Images have size 32x32x3 ➔ one neuron in hidden layer has 3072 weights!
  - With images of size 1024x1024x3 ➔ one neuron in hidden layer has 3,145,728 weights!
  - This explodes quickly if you increase the number of neurons & layers.
- Alternative: enforce local connectivity!

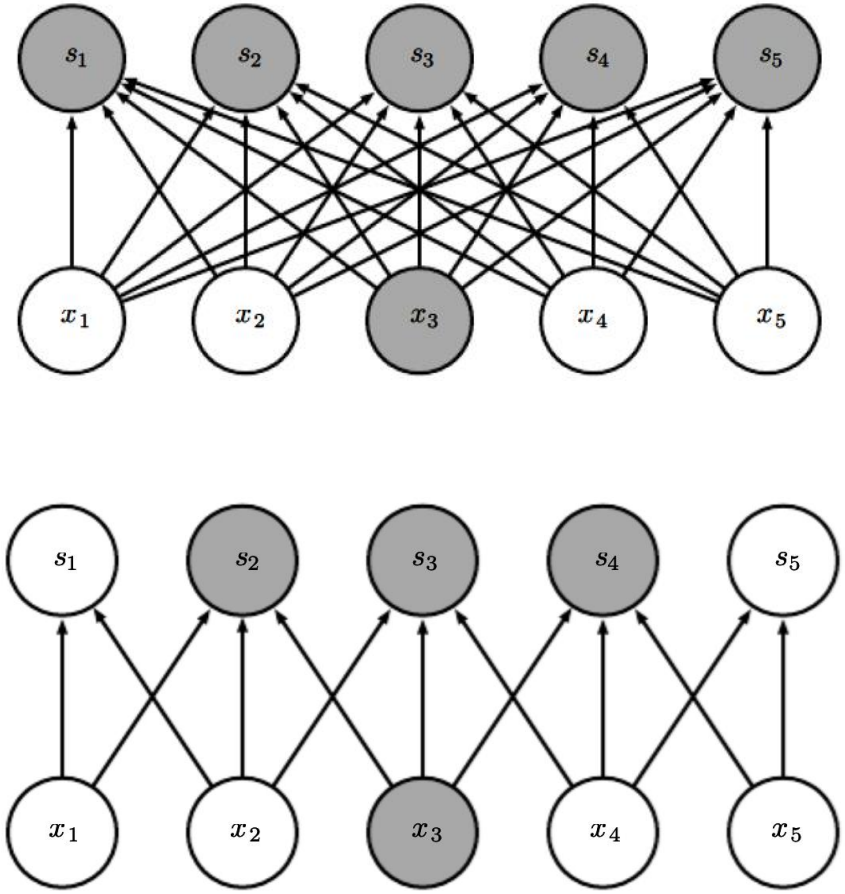
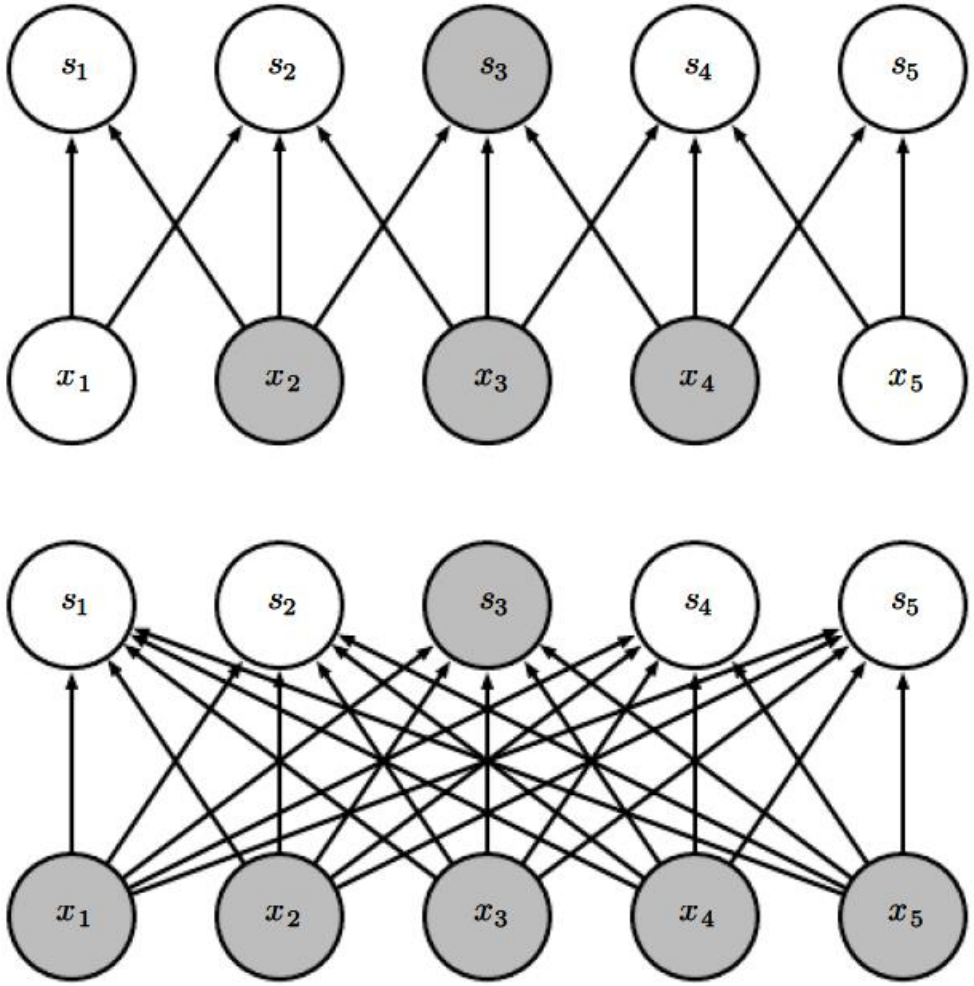


Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

CNNs vs. MLPs: Curse of Dimensionality



When things go deep, an output may depend on all or most of the input:

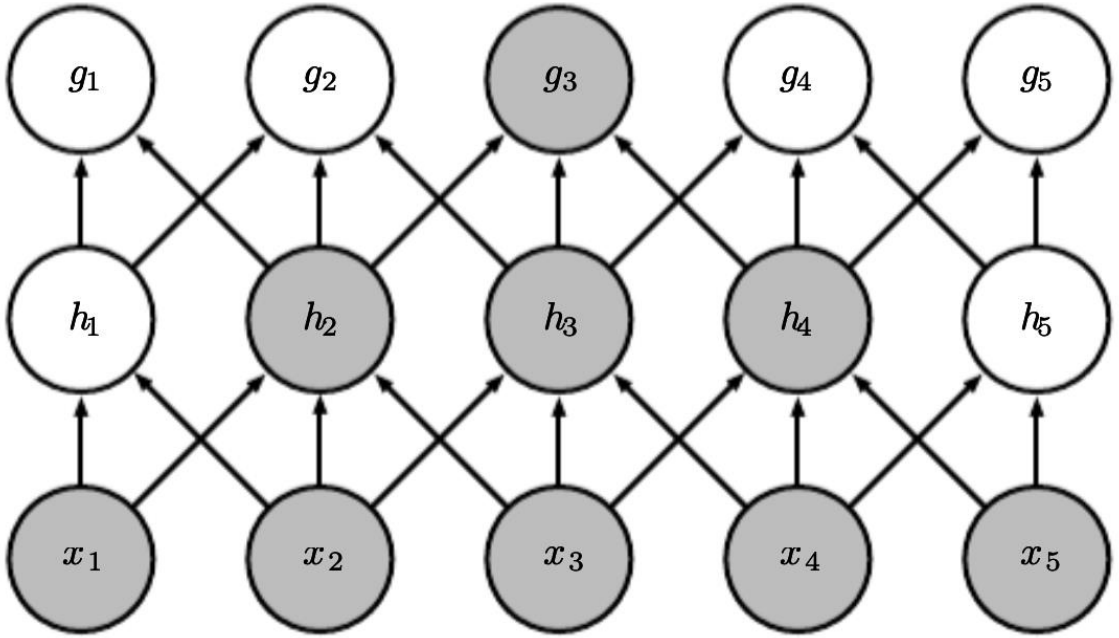


Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

How Many Samples are Needed to Learn a Convolutional Neural Network?

Simon S. Du*[1], Yining Wang*[1], Xiyu Zhai[2], Sivaraman Balakrishnan[3], Ruslan Salakhutdinov[1], and Aarti Singh[1]

[1]Machine Learning Department, Carnegie Mellon University
[2]University of Cambridge
[3]Department of Statistics, Carnegie Mellon University

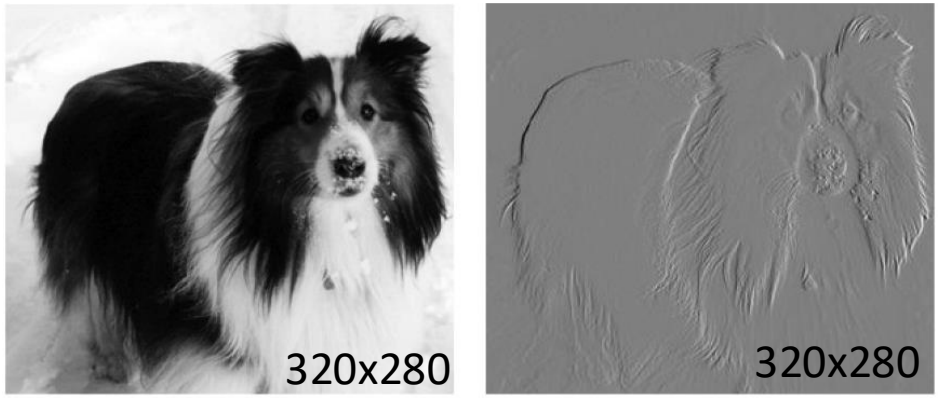
May 22, 2018

Abstract

A widespread folklore for explaining the success of convolutional neural network (CNN) is that CNN is a more compact representation than the fully connected neural network (FNN) and thus requires fewer samples for learning. We initiate the study of rigorously characterizing the sample complexity of learning convolutional neural networks. We show that for learning an m-dimensional convolutional filter with linear activation acting on a d-dimensional input, the sample complexity of achieving population prediction error of ϵ is $\widetilde{O}(m/\epsilon^2)$[1], whereas its FNN counterpart needs at least $\Omega(d/\epsilon^2)$ samples. Since $m \ll d$, this result demonstrates the advantage of using CNN. We further consider the sample complexity of learning a one-hidden-layer CNN with linear activation where both the m-dimensional convolutional filter and the r-dimensional output weights are unknown. For this model, we show the sample complexity is $\widetilde{O}\left((m+r)/\epsilon^2\right)$ when the ratio between the stride size and the filter size is a constant. For both models, we also present lower bounds showing our sample complexities are tight up to logarithmic factors. Our main tools for deriving these results are localized empirical process and a new lemma characterizing the convolutional structure. We believe these tools may inspire further developments in understanding CNN.

CNNs vs. MLPs: Curse of Dimensionality

- Parameter sharing
  - In regular ANN, each weight is independent
- In CNN, a layer might re-apply the same convolution and therefore, share the parameters of a convolution
  - Reduces storage and learning time



- For a neuron in the next layer:
  - With ANN: 320x280x320x280 multiplications
  - With CNN: 320x280x3x3 multiplications

Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

CNNs vs. MLPs: Equivariance

- Equivariant to translation
  - The output will be the same, just translated, since the weights are shared.

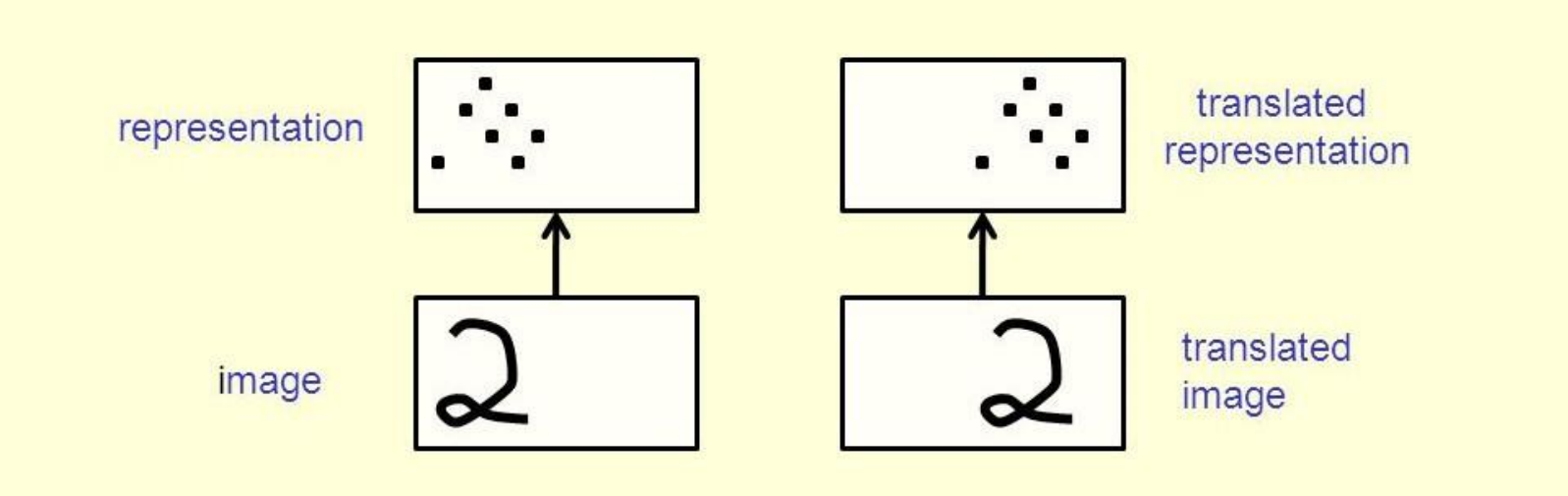
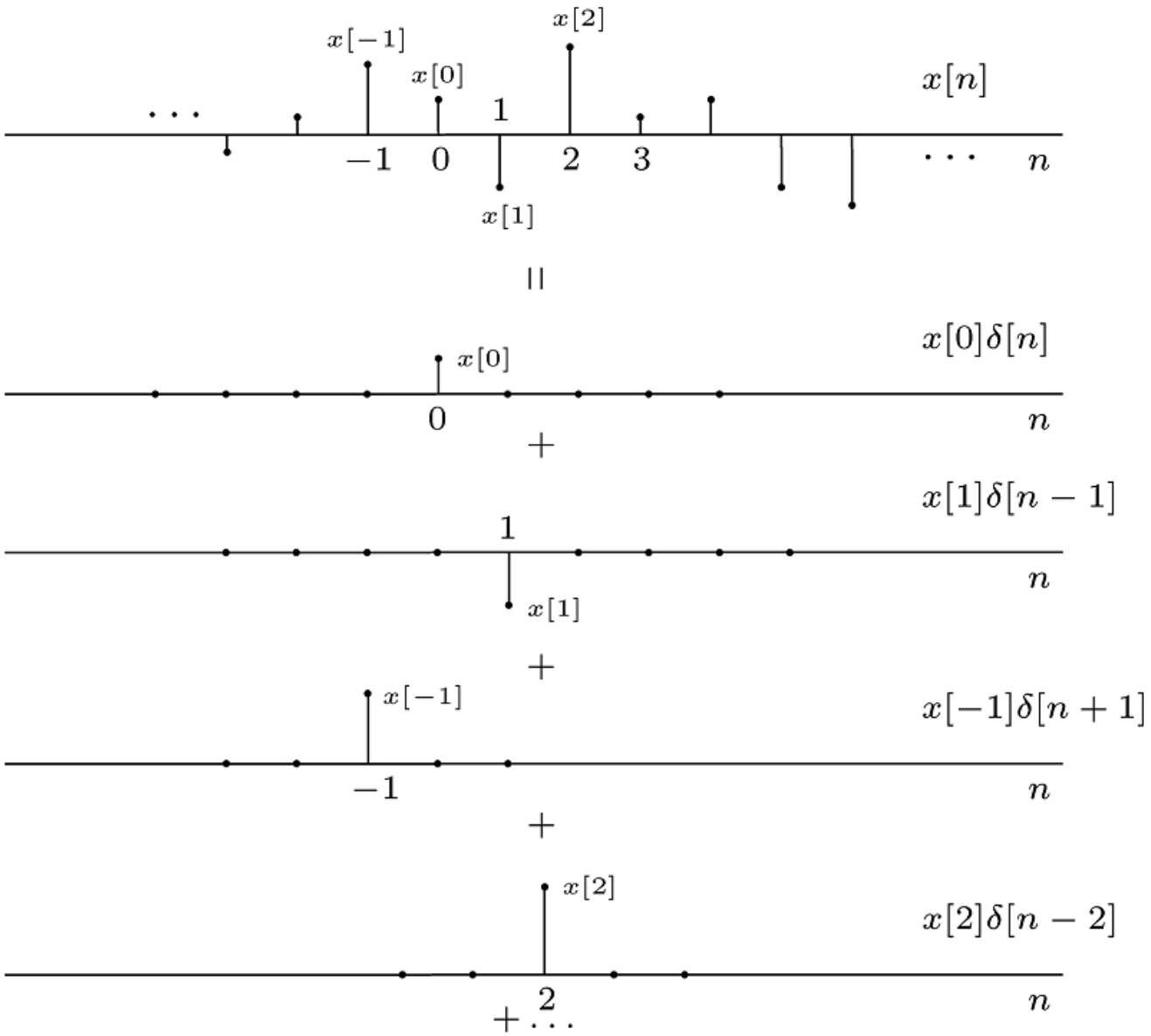


Figure: https://towardsdatascience.com/translational-invariance-vs-translational-equivariance-f9fbc8fca63a

- Not equivariant to scale or rotation.

A crash course on Convolution

Formulating Signals in Terms of Impulse Signal



Alan V. Oppenheim and Alan S. Willsky

Formulating Signals in Terms of Impulse Signal

$$x[n] = \ldots + x[-2]\delta[n+2] + x[-1]\delta[n+1] + x[0]\delta[n] + x[1]\delta[n-1] + \ldots$$

$$\Downarrow$$

$$x[n] = \sum_{k=-\infty}^{+\infty} \underbrace{x[k]}_{\text{Coefficients}} \underbrace{\delta[n-k]}_{\text{Basic Signals}}$$

Important to note the "-" sign

Alan V. Oppenheim and Alan S. Willsky

Unit Sample Response

- Now suppose the system is **LTI**, and define the *unit sample response* $h[n]$:

$$\delta[n] \longrightarrow h[n]$$

$$\Downarrow$$

From Time-Invariance:

$$\delta[n-k] \longrightarrow h[n-k]$$

From Linearity:

$$x[n] = \sum_{k=-\infty}^{+\infty} x[k]\,\delta[n-k] \longrightarrow y[n] = \underbrace{\sum_{k=-\infty}^{+\infty} x[k]\,h[n-k]}_{convolution\ sum} = x[n] * h[n]$$

Alan V. Oppenheim and Alan S. Willsky

Conclusion

The output of *any* DT LTI System is a convolution of the input signal with the unit-sample response, *i.e.*

$$\text{Any DT LTI} \longleftrightarrow y[n] = x[n] * h[n]$$

$$= \sum_{k=-\infty}^{+\infty} x[k]\, h[n-k]$$

As a result, any DT LTI Systems are *completely characterized* by its unit sample response

Alan V. Oppenheim and Alan S. Willsky

Power of convolution

- Describe a “system” (or operation) with a very simple function (impulse response).
- Determine the output by convolving the input with the impulse response

Convolution

- Definition of continuous-time convolution

$$x(t) * h(t) = \int x(\tau)h(t-\tau)\,d\tau$$

$$h(\tau) \xrightarrow{Flip} h(-\tau) \xrightarrow{Slide} h(t-\tau) \xrightarrow{Multiply}$$

$$x(\tau)h(t-\tau) \xrightarrow{Integrate} \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau$$

Alan V. Oppenheim and Alan S. Willsky

Convolution

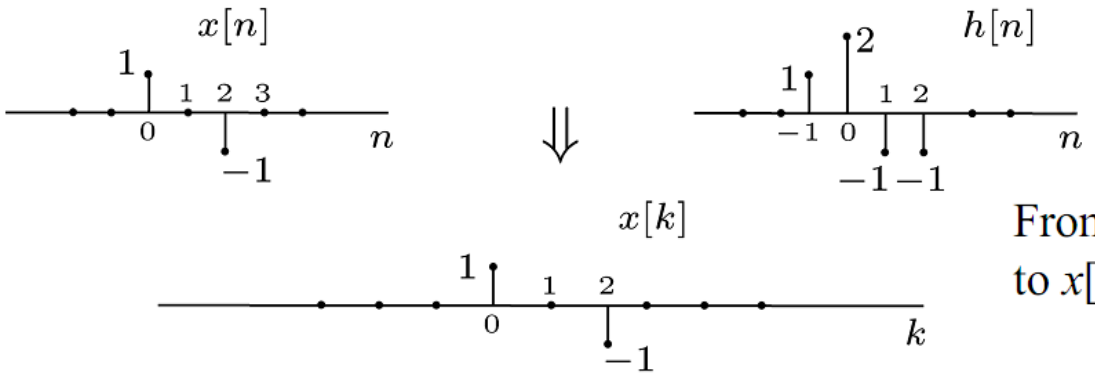
- Definition of discrete-time convolution

$$x[n] * h[n] = \sum x[k]h[n-k]$$

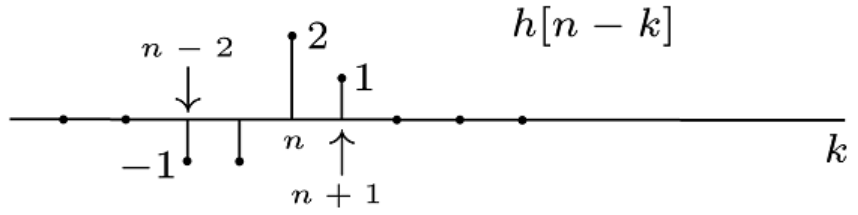
Choose the value of n and consider it fixed

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k]$$

View as functions of k with n fixed



From $x[n]$ and $h[n]$ to $x[k]$ and $h[n-k]$



Alan V. Oppenheim and Alan S. Willsky

Discrete-time 2D Convolution

- For images, we need two-dimensional convolution:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n]$$

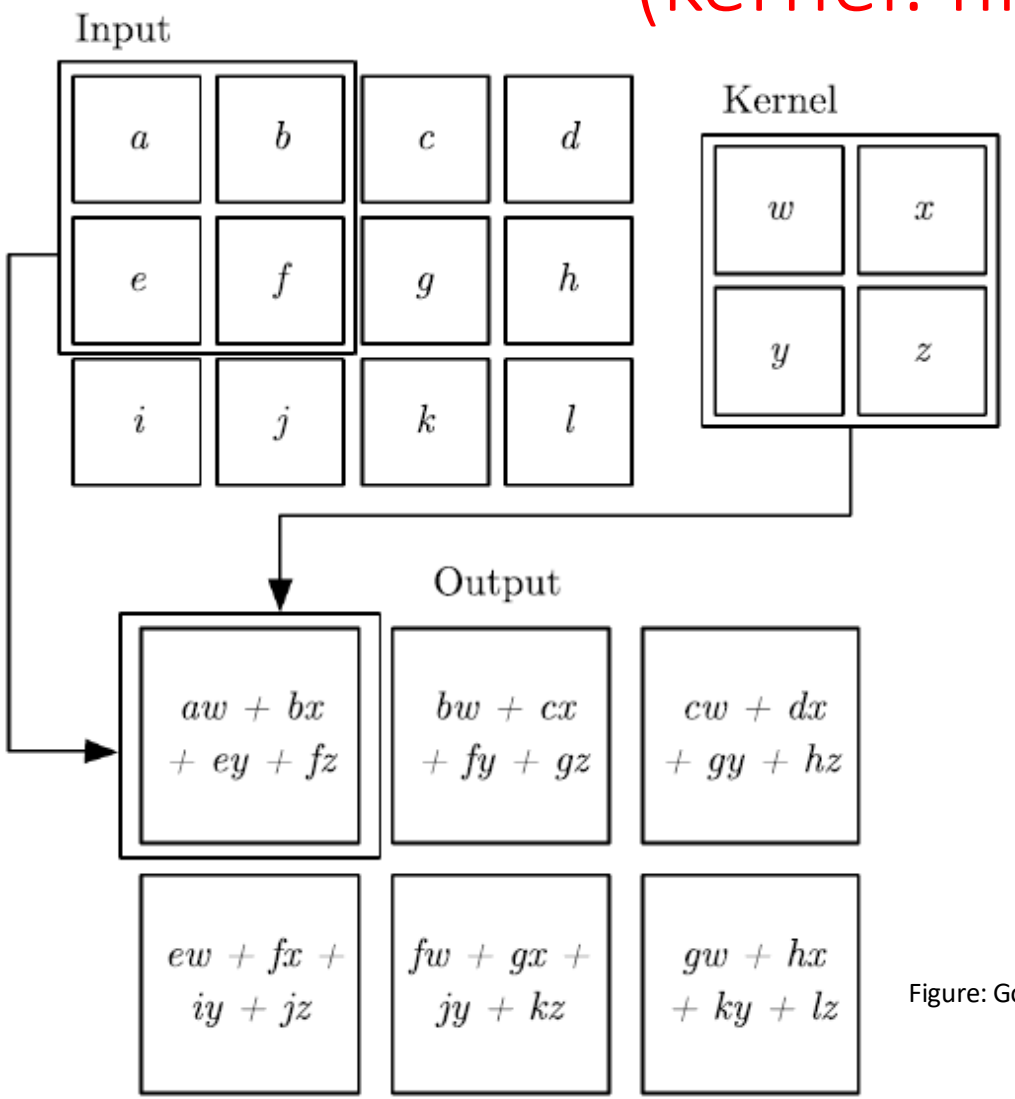
- These multi-dimensional arrays are called tensors
- We have commutative property:

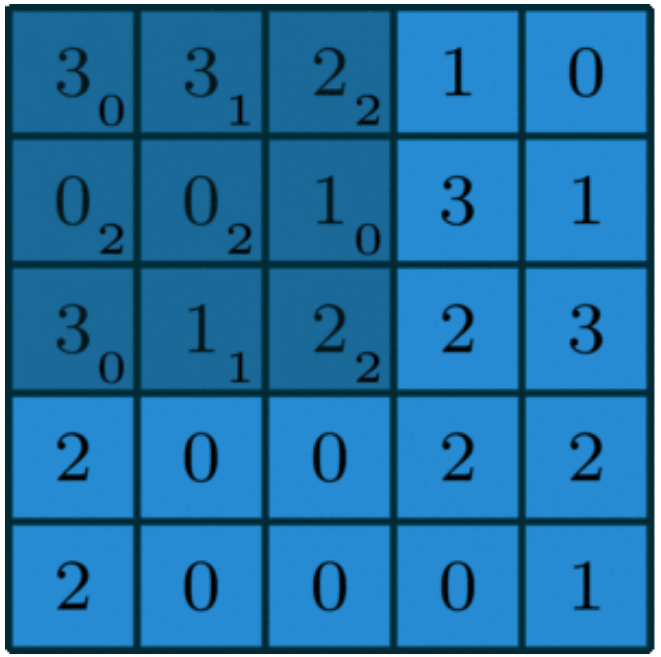
$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n]$$

- Instead of subtraction, we can also write (easy to drive by a change of variables). This is called cross-correlation:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i+m, j+n]K[m,n]$$

Example multi-dimensional convolution
(kernel: finite impulse response)





https://github.com/vdumoulin/conv_arithmetic

Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

What can filters do?
Rectangular filter

g[m,n]

$\otimes$

h[m,n]

=

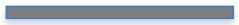
f[m,n]

Slide: A. Torralba

What can filters do?
Rectangular filter

g[m,n]

$\otimes$



h[m,n]

=

f[m,n]

Slide: A. Torralba

What can filters do?
Rectangular filter

g[m,n]

$\otimes$

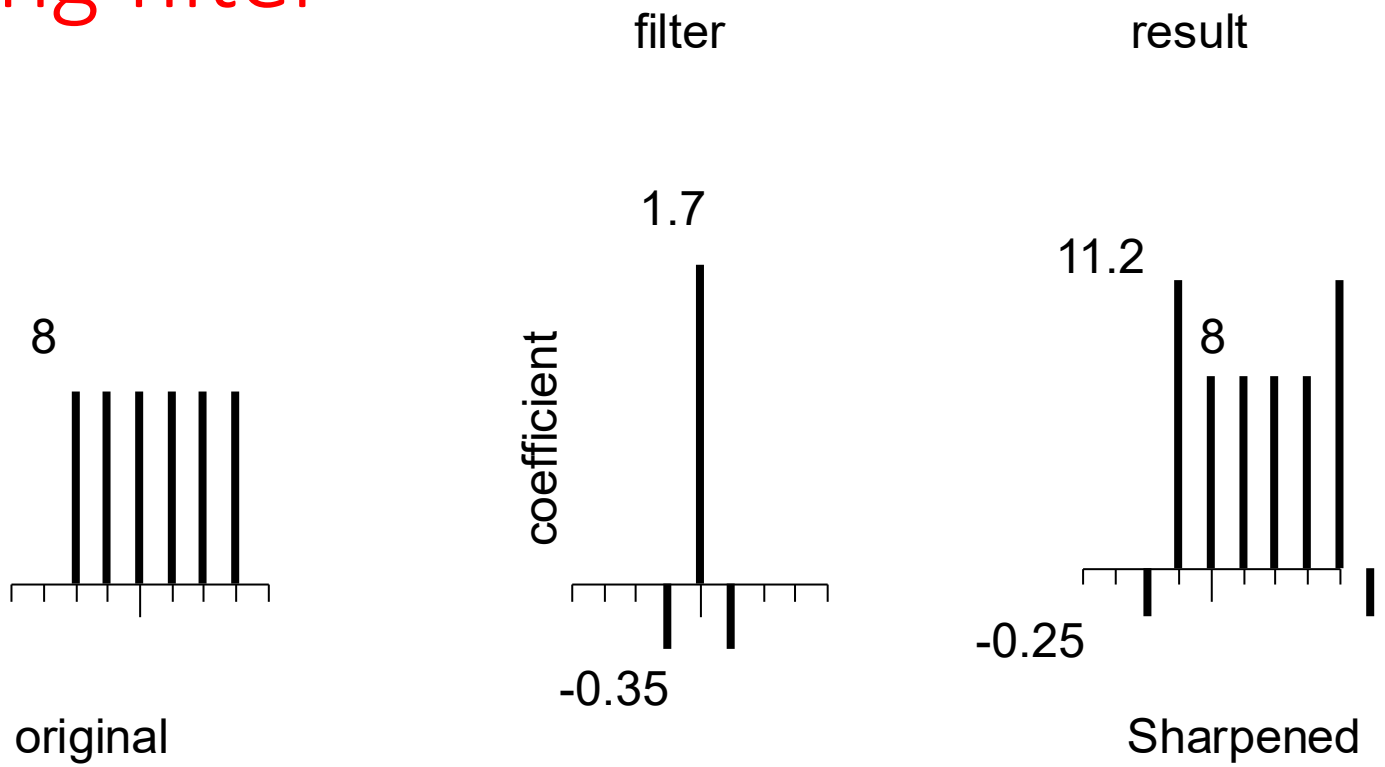
h[m,n]

=

f[m,n]

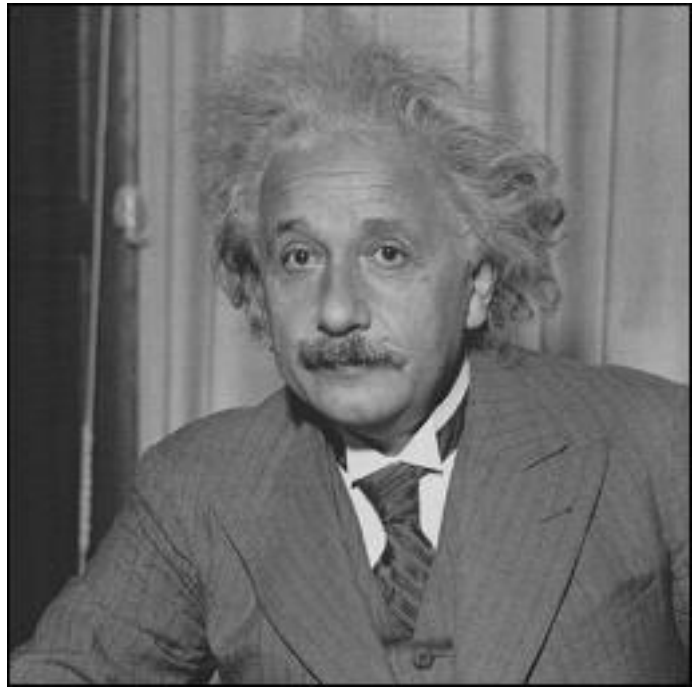
Slide: A. Torralba

What can filters do?
Sharpening filter

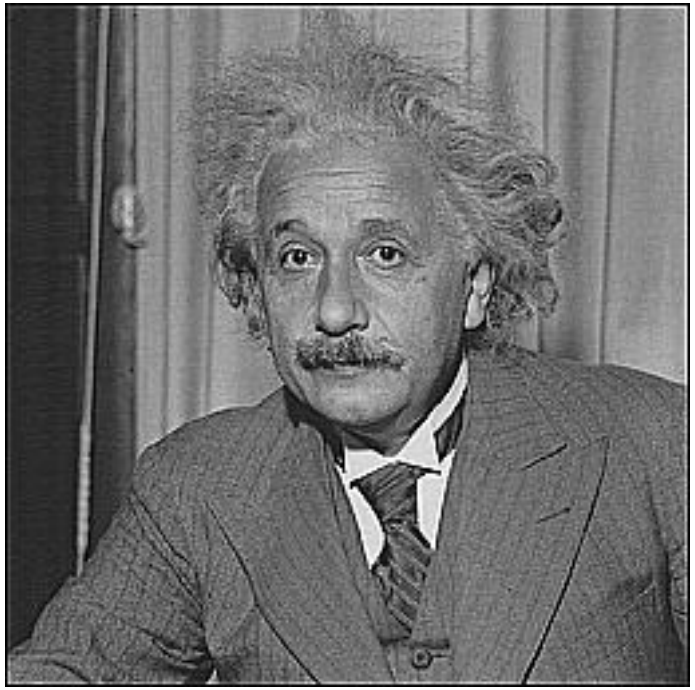


Slide: A. Torralba

What can filters do?
Sharpening filter



before

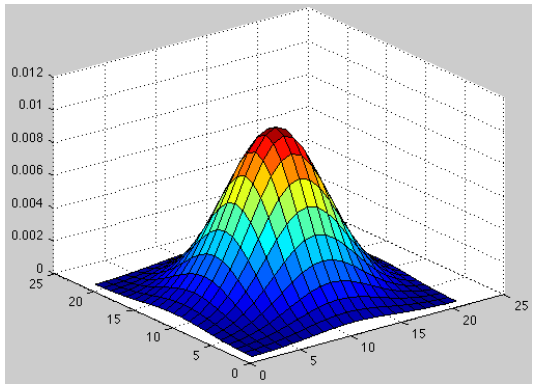


after

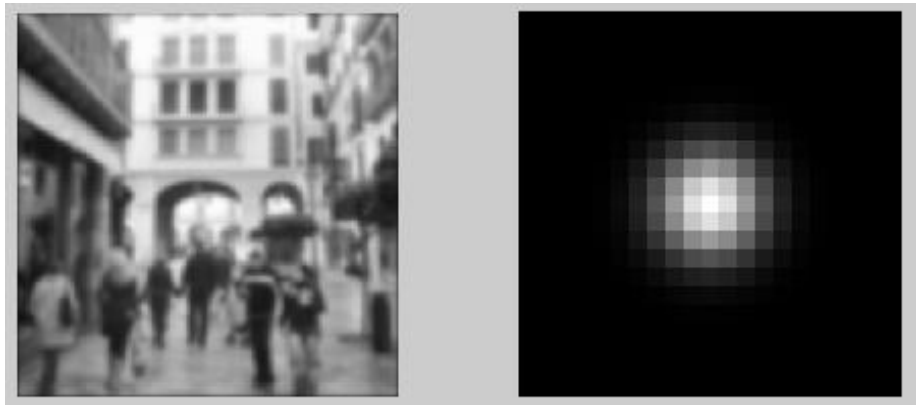
Slide: A. Torralba

What can filters do? Gaussian filter

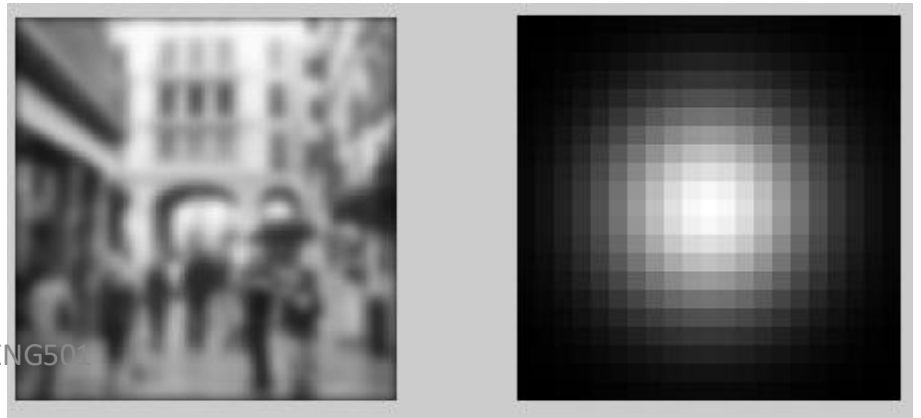
$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



σ=1



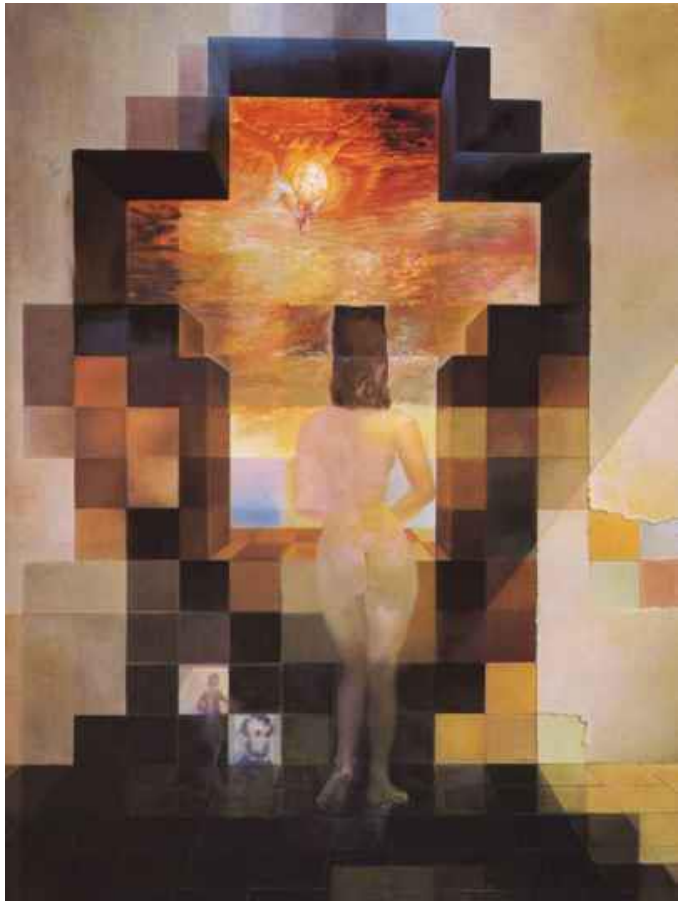
σ=2



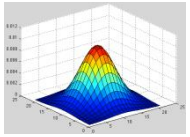
σ=4

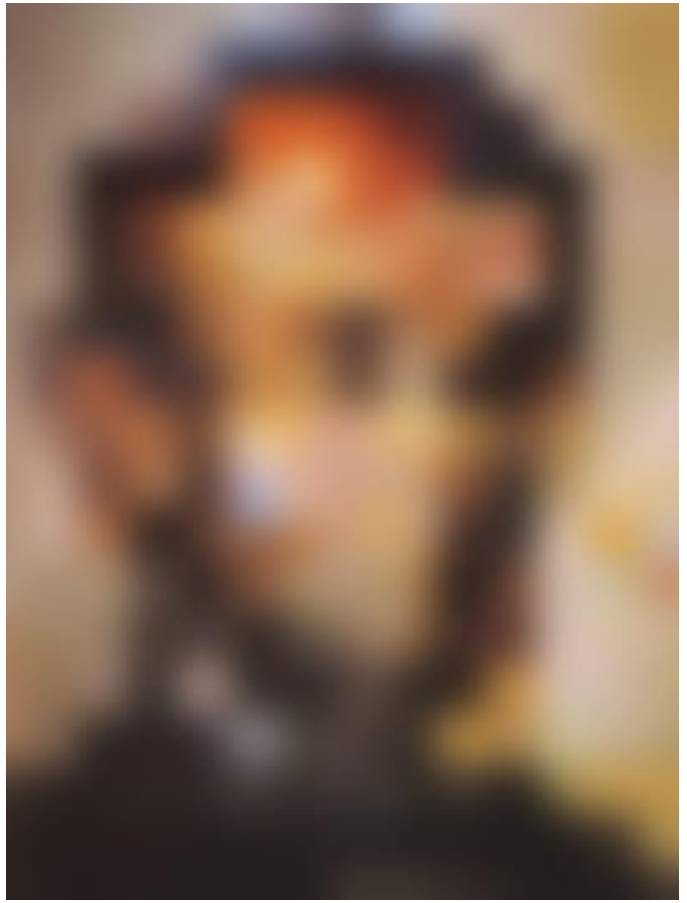
Slide: A. Torralba

Global to Local Analysis



Dali





Slide: A. Torralba

What can filters do?
[-1 1]

g[m,n]

$\otimes$

[-1, 1]

h[m,n]

=

f[m,n]

Slide: A. Torralba

What can filters do?
$[-1\ 1]^T$

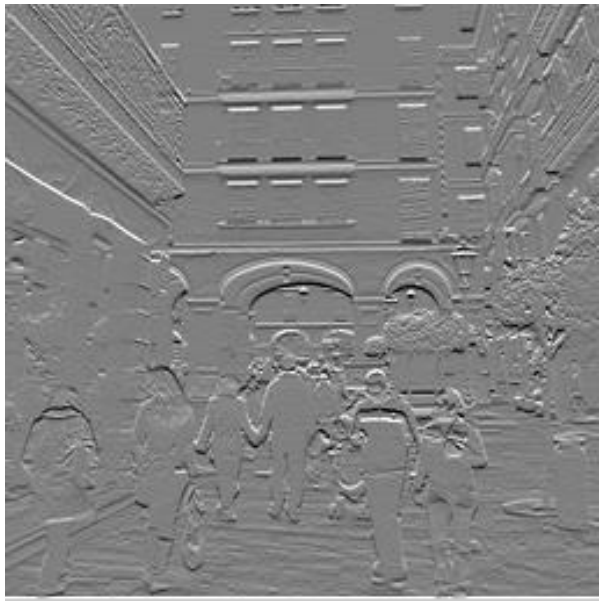
g[m,n]

$\otimes$

$[-1, 1]^T$

h[m,n]

=



f[m,n]

Slide: A. Torralba

Overview of CNN

CNN layers

- Operations in a CNN:
  - Convolution (in parallel) to produce pre-synaptic activations
  - Detector: Non-linear function
  - Pooling: A summary of a neighborhood

- Pooling of a region in a feature/activation map:
  - Max
  - Average
  - L2 norm
  - Weighted average acc. to the distance to the center
  - …

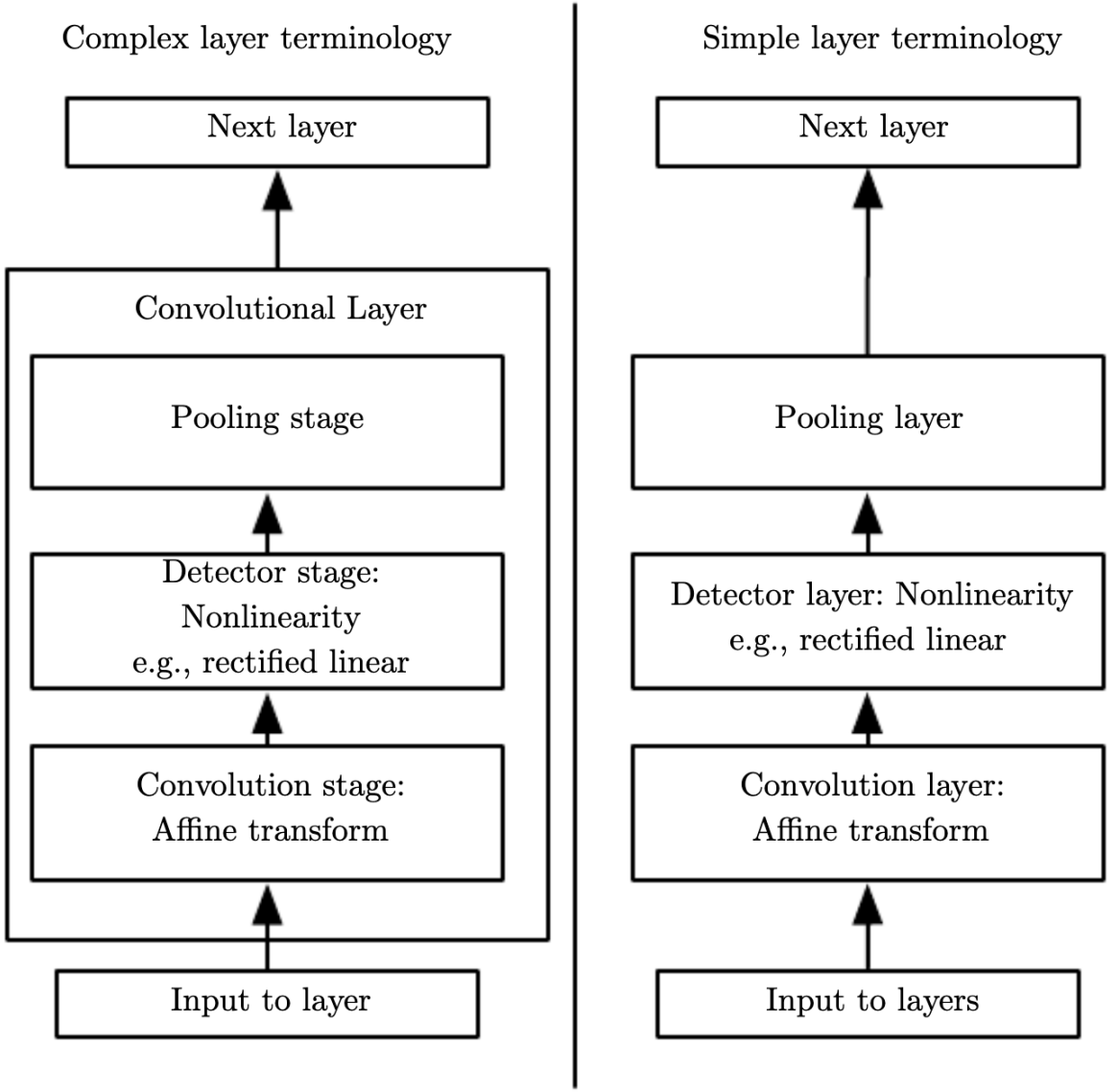
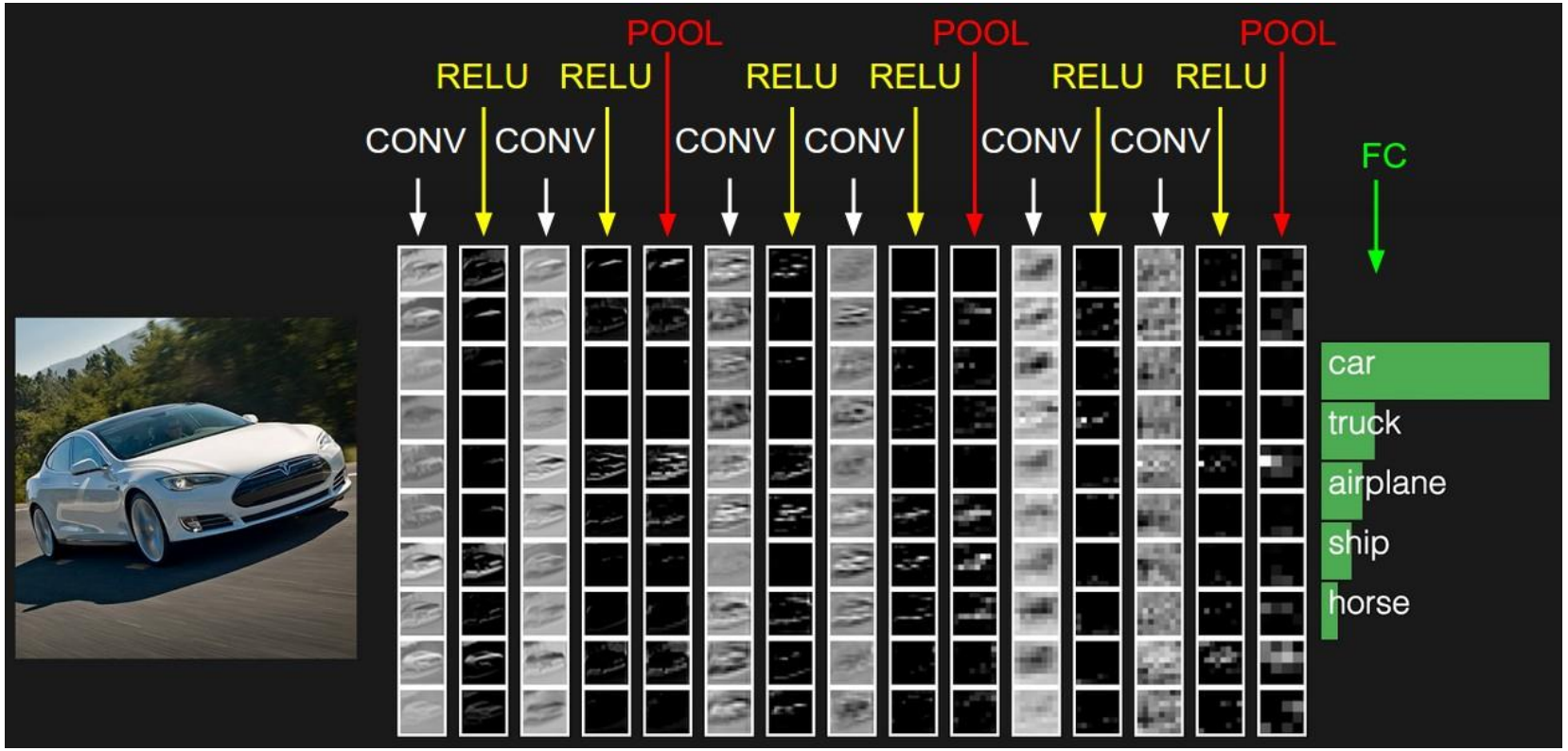


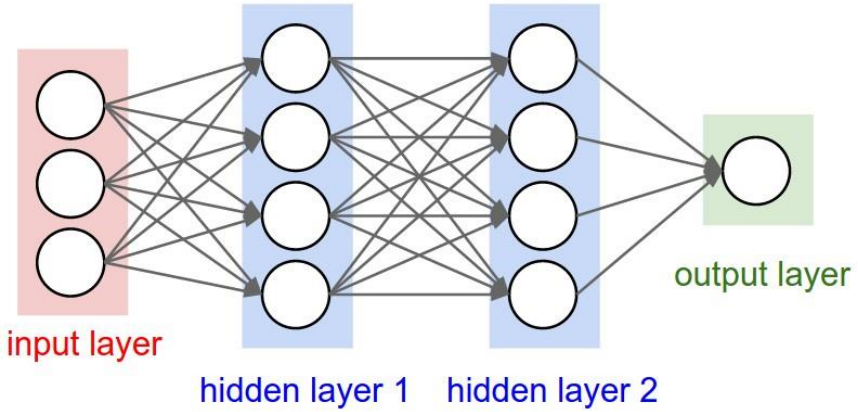
Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

An example architecture

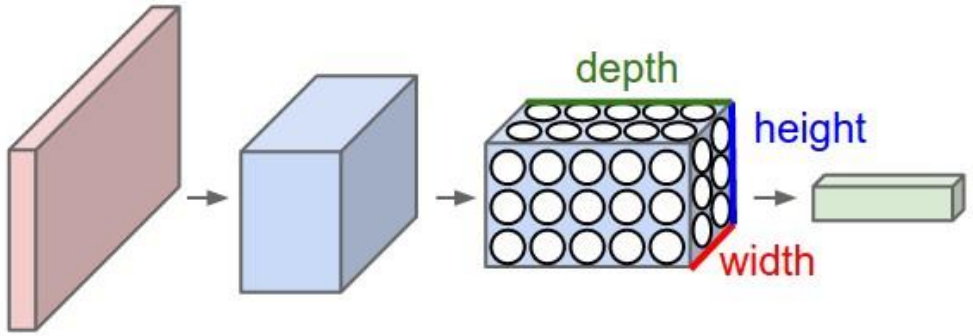


http://cs231n.github.io/convolutional-networks/

Regular ANN



CNN



http://cs231n.github.io/convolutional-networks/

OPERATIONS IN A CNN: Convolution

Convolution in CNN

- The weights correspond to the kernel
- The weights are shared in a channel (depth slice)
- We are effectively learning filters that respond to some part/entities/visual-cues etc.

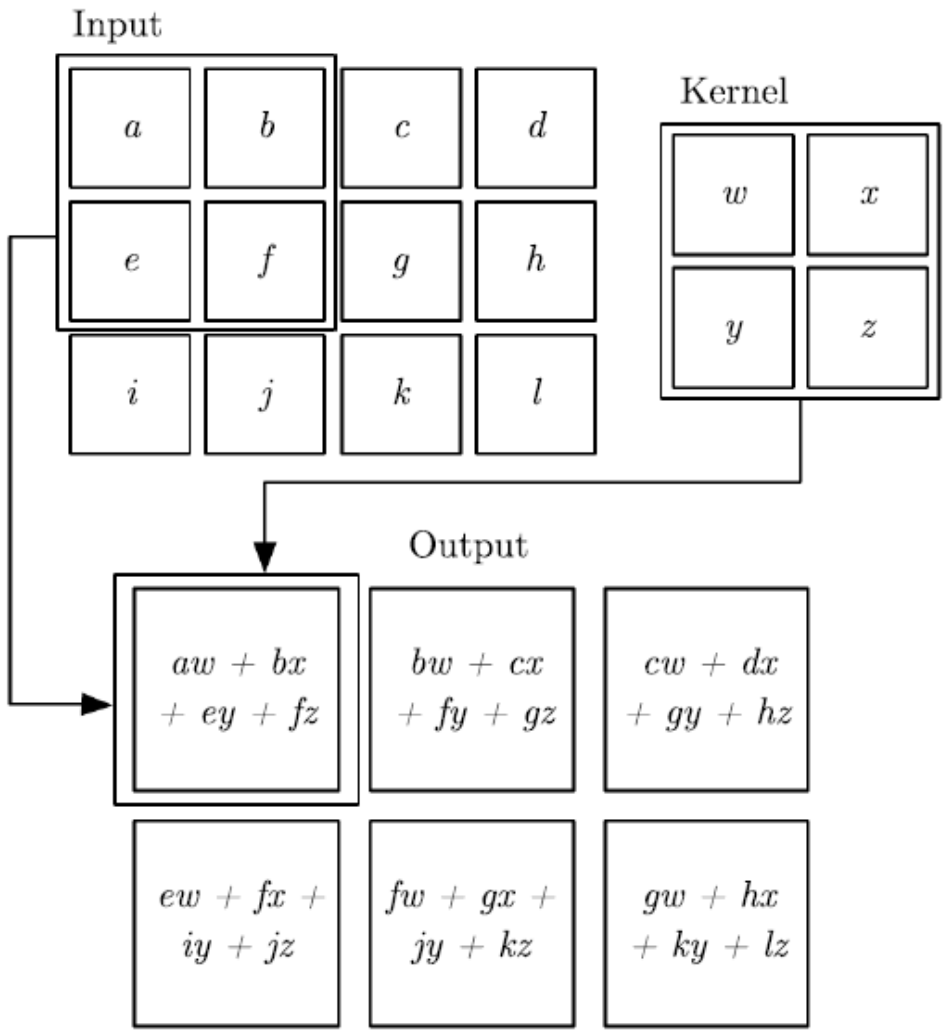


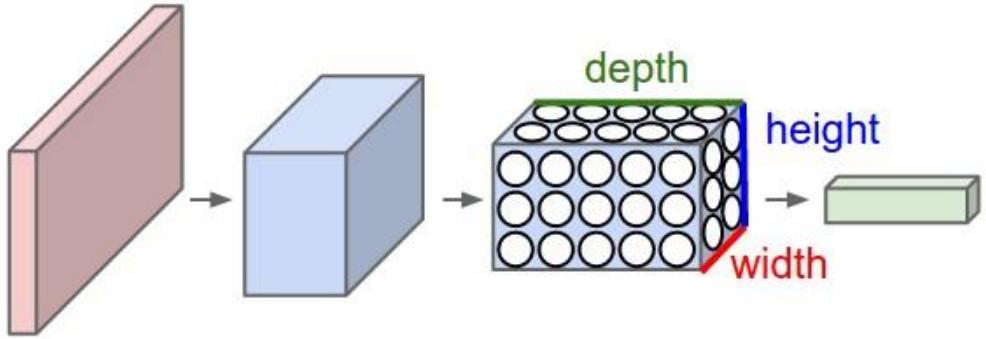
Figure: Goodfellow et al., "Deep Learning", MIT Press, 2016.

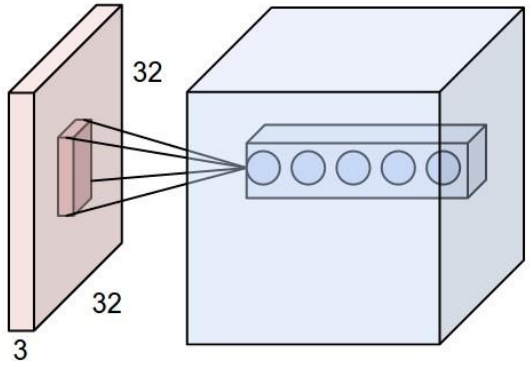
Local connectivity in CNN
= Receptive fields

- Each neuron is connected to only a local neighborhood, i.e., receptive field
- The size of the receptive field ➔ another hyper-parameter.

Connectivity in CNN

- Local: The behavior of a neuron does not change other than being restricted to a subspace of the input.
- Each neuron is connected to slice of the previous layer
- A layer is actually a volume having a certain width x height and depth (or channel)
- A neuron is connected to a subspace of width x height but to all channels (depth)
- Example: CIFAR-10
  - Input: 32 x 32 x 3 (3 for RGB channels)
  - A neuron in the next layer with receptive field size 5x5 has input from a volume of 5x5x3.

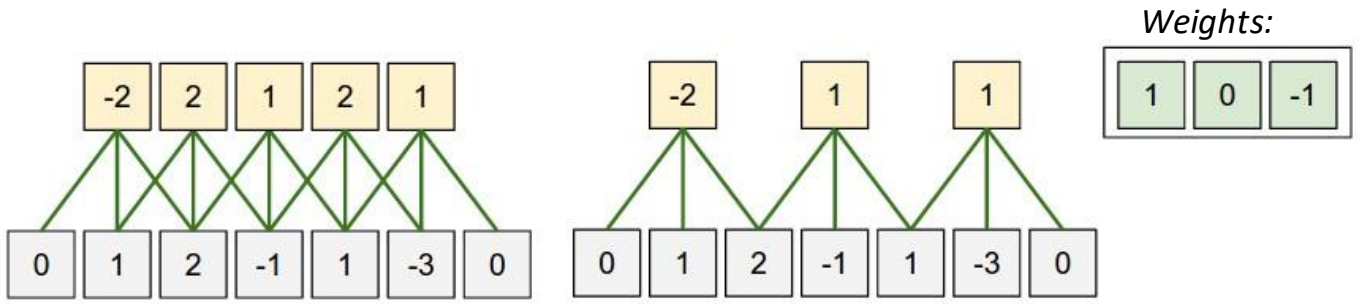




http://cs231n.github.io/convolutional-networks/

Important parameters

- Depth (number of channels)
  - We will have more neurons getting input from the same receptive field
  - This is similar to the hidden neurons with connections to the same input
  - These neurons learn to become selective to the presence of different signals in the same receptive field
- Stride
  - The amount of space between neighboring receptive fields
  - If it is small, RFs overlap more
  - It it is big, RFs overlap less
- How to handle the boundaries?
  i. Option 1: Don't process the boundaries. Only process pixels on which convolution window can be placed fully.
  ii. Option 2: Zero-pad the input so that convolution can be performed at the boundary pixels.



http://cs231n.github.io/convolutional-networks/

Padding illustration

- Only convolution layers are shown.
- Top: no padding ➔ layers shrink in size.
- Bottom: zero padding ➔ layers keep their size fixed.

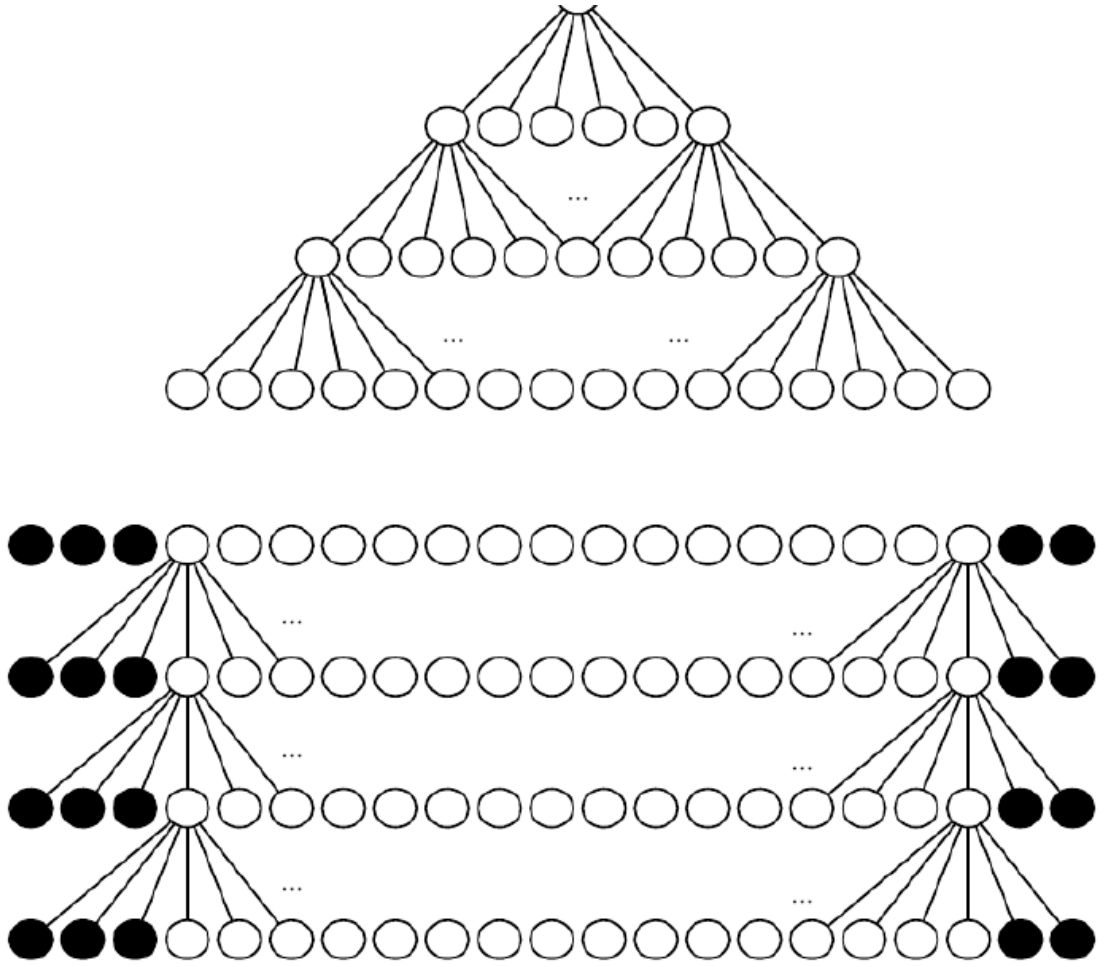


Figure 9.11: *The effect of zero padding on network size*: Consider a convolutional network with a kernel of width six at every layer. In this example, do not use any pooling, so only the convolution operation itself shrinks the network size. *Top)* In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive and some shrinking is inevitable in this kind of architecture. *Bottom)* By adding five implicit zeroes to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

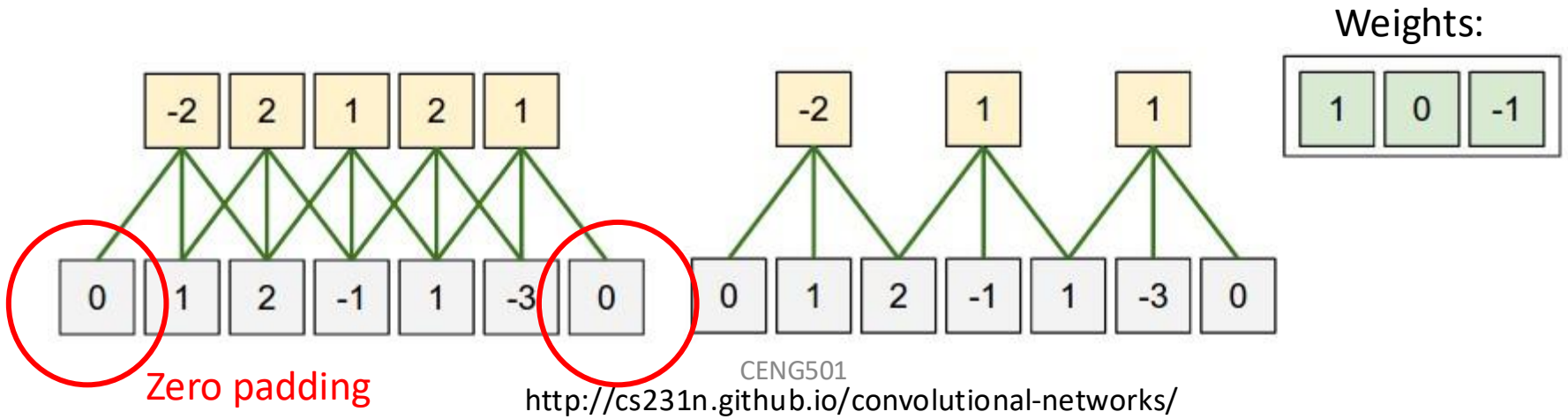
Figure: Goodfellow et al., “Deep Learning”, MIT Press, 2016.

Size of the next layer

- Along a dimension:
  - W: Size of the input
  - F: Size of the receptive field
  - S: Stride
  - P: Amount of zero-padding
- Then: the number of neurons as the output of a convolution layer:

$$\frac{W - F + 2P}{S} + 1$$

- If this number is not an integer, your strides are incorrect and your neurons cannot tile nicely to cover the input volume



http://cs231n.github.io/convolutional-networks/

Size of the next layer

- Arranging these hyperparameters can be problematic
- Example:
- If W=10, P=0, and F=3, then

$$\frac{W - F + 2P}{S} + 1 = \frac{10 - 3 + 0}{S} + 1 = \frac{7}{S} + 1$$

i.e., S cannot be an integer other than 1 or 7.

- Zero-padding is your friend here.

Real example – AlexNet (Krizhevsky et al., 2012)

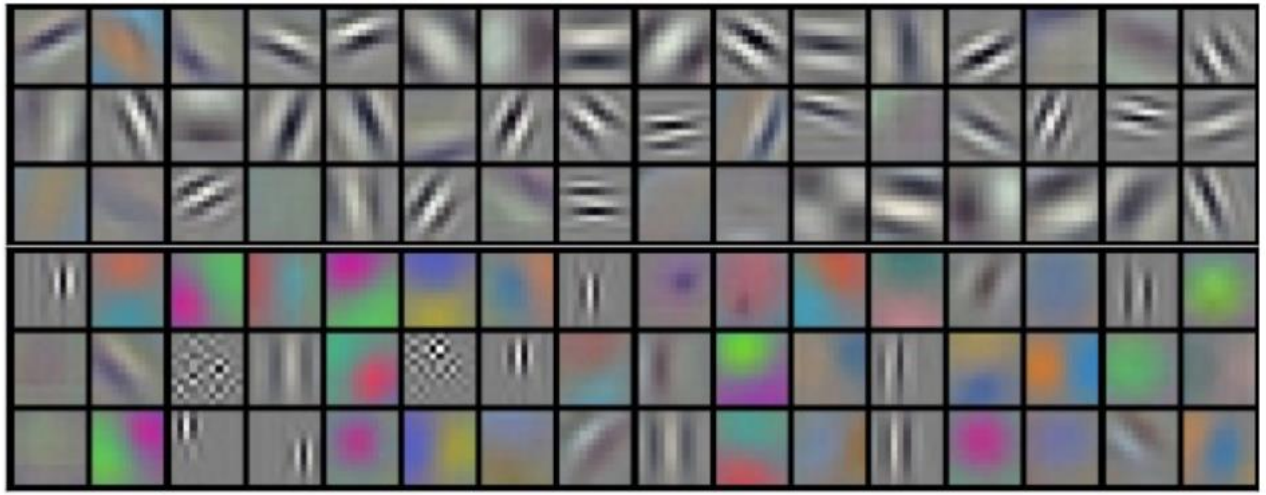
- Image size: 227×227×3
- W=227, F=11, S=4, P=0 ➔ $\frac{227-11}{S}+1=55$

  (55 => the width of the convolution layer)
- Convolution layer: 55×55×96 neurons

  (96: the depth, the number of channels)
- Therefore, the first layer has 55×55×96 = 290,400 neurons
  - Each has 11×11×3 receptive field ➔ 363 weights and 1 bias
  - Then, 290,400×364 = 105,705,600 parameters just for the first convolution layer (if there were no weight sharing)
  - With weight sharing: 96 x 364 = 34,944

Real example – AlexNet (Krizhevsky et al., 2012)

- However, we can share the parameters
  - For each channel (slice of depth), have the same set of weights
  - If 96 channels, this means 96 different set of weights
  - Then, 96×364 = 34,944 parameters
  - 364 weights shared by 55×55 neurons in each channel



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

http://cs231n.github.io/convolutional-networks/

More on connectivity

Small RF & Stacking

- E.g., 3 CONV layers of 3x3 RFs
- Pros:
  - Same extent for these example figures
  - With non-linearity added on 2nd and 3rd layers ➔ More expressive! More representational capacity!
  - Less parameters:

    3 layers x [(3 x 3 x C) x C] = 27CxC
- Cons?

Large RF & Single Layer

- 7x7 RFs of single CONV layer
- Pros?
- Cons:
  - One layer => Linear capacity
  - More parameters:

    (7x7xC)xC = 49CxC

So, we prefer a stack of small filter sizes against big ones

http://cs231n.github.io/convolutional-networks/

Implementation Details: NumPy example

- Suppose input is `X` of shape (11,11,4)
- Depth slice at depth d (i.e., channel d): `X[:,:,d]`
- Depth column at position (x,y): `X[x,y,:]`
- F: 5, P:0 (no padding), S=2
  - Output volume (V) width, height = (11-5+0)/2+1 = 4
- Example computation for some neurons in first channel:

```
V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0
V[1,0,0] = np.sum(X[2:7,:5,:] * W0) + b0
V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0
V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0
```

- Note that this is just along one dimension (x)

http://cs231n.github.io/convolutional-networks/

Implementation Details: NumPy example

- A second activation map (channel):

```
V[0,0,1] = np.sum(X[:5,:5,:] * W1) + b1
V[1,0,1] = np.sum(X[2:7,:5,:] * W1) + b1
V[2,0,1] = np.sum(X[4:9,:5,:] * W1) + b1
V[3,0,1] = np.sum(X[6:11,:5,:] * W1) + b1
V[0,1,1] = np.sum(X[:5,2:7,:] * W1) + b1
V[2,3,1] = np.sum(X[4:9,6:11,:] * W1) + b1
```

(example of going along y)

(or along both)

http://cs231n.github.io/convolutional-networks/

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

http://cs231n.github.io/convolutional-networks/