

CENG501 – Deep Learning

Week 3

Fall 2024

Sinan Kalkan

Dept. of Computer Engineering, METU

Previously on CENG501

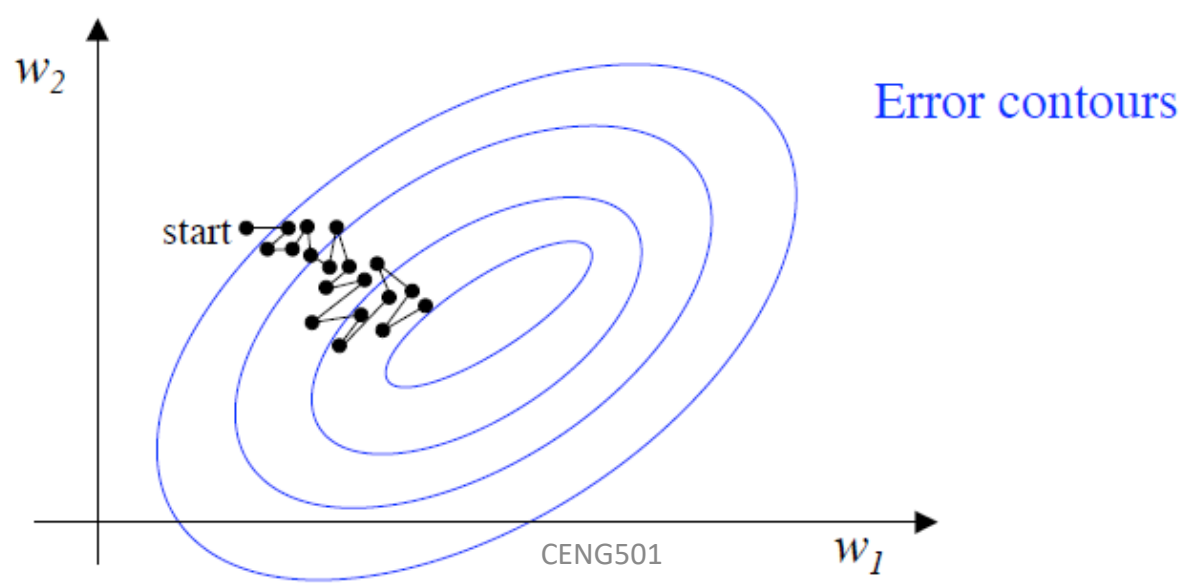
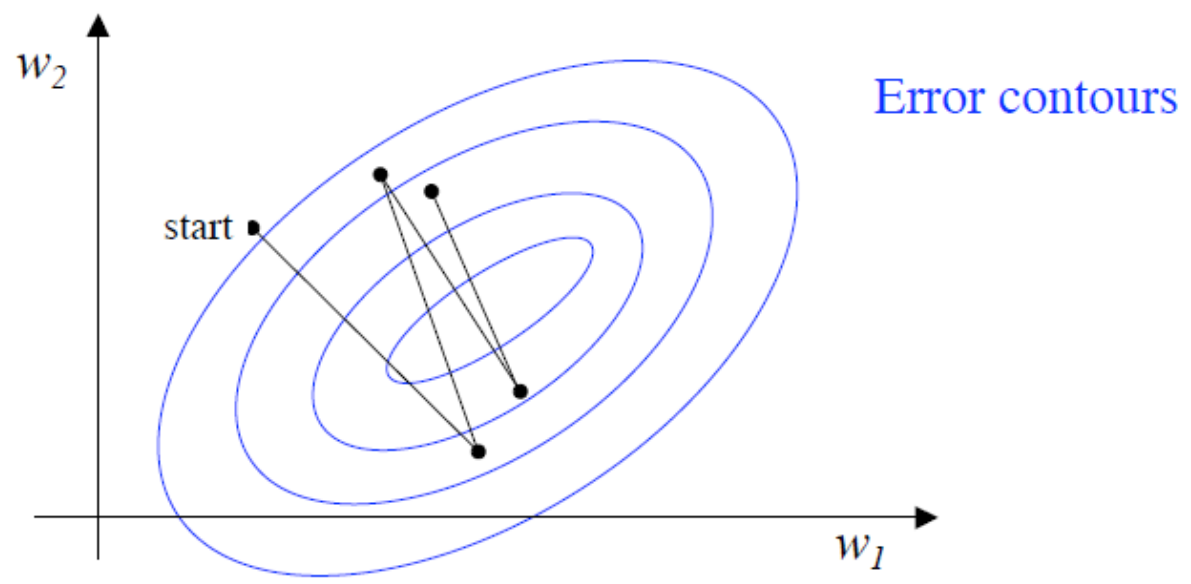
Topics covered

- Loss Functions
- Activation Functions
- Optimization Perspective
- Challenges of the Loss Surface
- **Setting the Learning Rate**

Previously on CENG501

Setting the learning rate

Previously on CENG501



Alternatives

Previously on CENG501

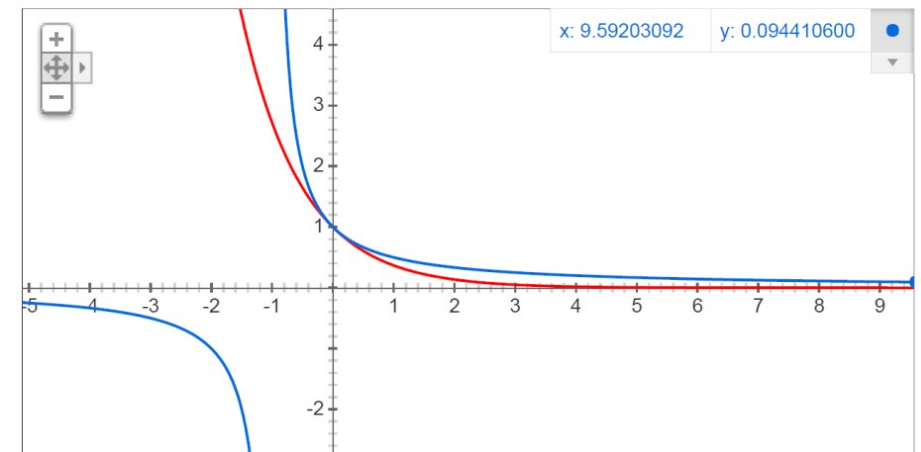
- Single global learning rate
 - Constant Learning Rate
 - Scheduling Learning Rate
- Per-parameter learning rate
 - AdaGrad
 - RMSprop
 - Adam
 - AdaDelta

Previously on CENG501

Global Methods: Scheduling the learning rate

- Step decay
 - $\eta' \leftarrow \eta \times c$, where c could be 0.5, 0.4, 0.3, 0.2, 0.1 etc.
- Exponential decay:
 - $\eta = \eta_0 e^{-kt}$, where t is iteration number
 - η_0, k : hyperparameters
- $1/t$ decay:
 - $\eta = \eta_0 / (1 + kt)$
- If you have time, keep decay small and train longer

Graph for $1/(1+x)$, e^{-x}



Global Methods: warm-up

- Start with a small learning rate [1]
 - Constant learning rate
 - Gradually increasing
- Why? The first steps of learning appear to be very critical [2]

[1] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677. (this is not the first paper to do so)

[2] Achille, A., Rovere, M., & Soatto, S. (2018). Critical learning periods in deep networks. In International Conference on Learning Representations.

Global Methods: Cyclic Learning Rates

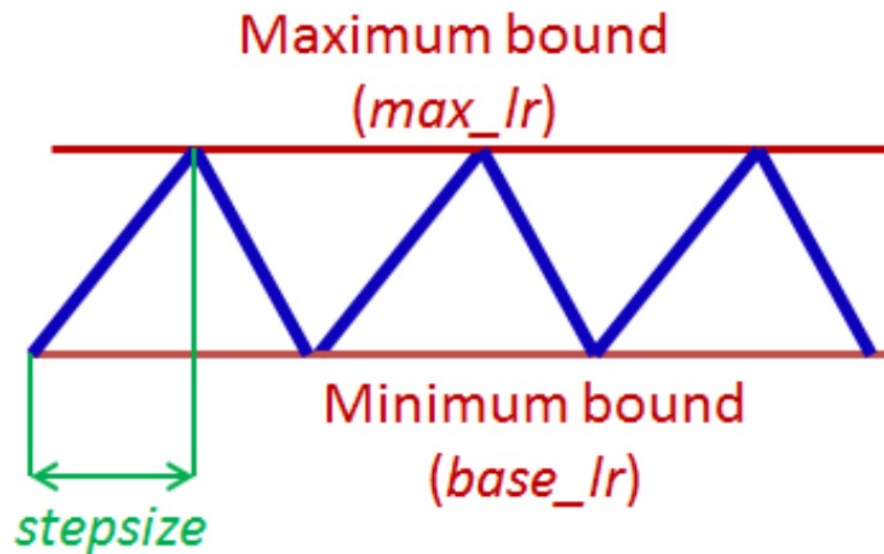


Figure 2. Triangular learning rate policy. The blue lines represent learning rate values changing between bounds. The input parameter *stepsize* is the number of iterations in half a cycle.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In 2017 IEEE winter conference on applications of computer vision (WACV) (pp. 464-472). IEEE.

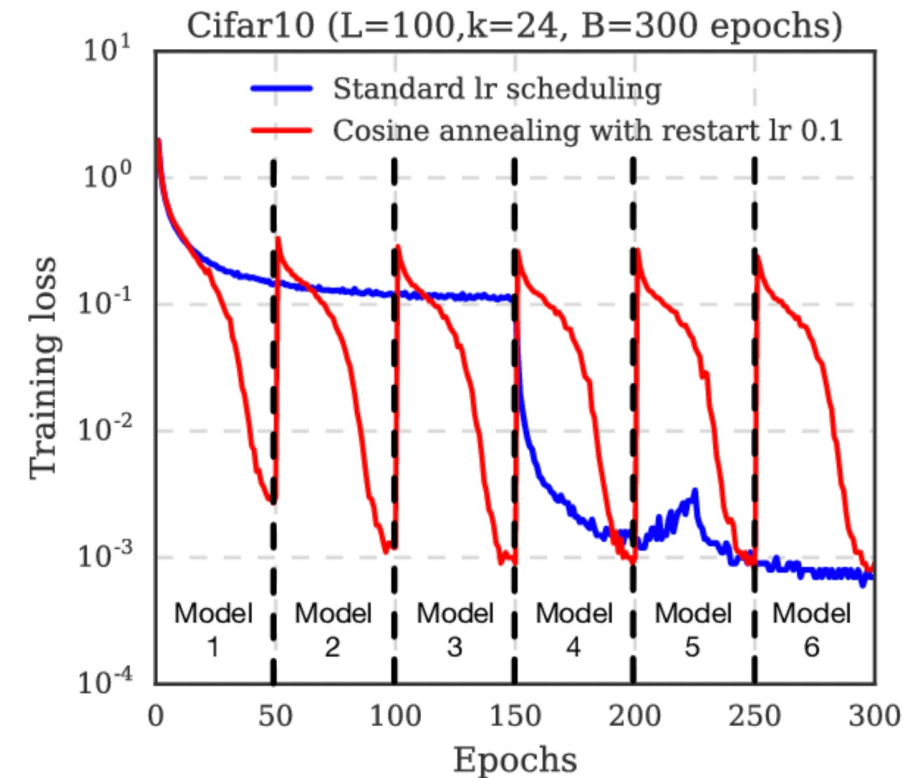
Global Methods: Cosine Scheduling

Cosine Annealing is a type of learning rate schedule that has the effect of starting with a large learning rate that is relatively rapidly decreased to a minimum value before being increased rapidly again. The resetting of the learning rate acts like a simulated restart of the learning process and the re-use of good weights as the starting point of the restart is referred to as a "warm restart" in contrast to a "cold restart" where a new set of small random numbers may be used as a starting point.

$$\eta_t = \eta_{min}^i + \frac{1}{2} (\eta_{max}^i - \eta_{min}^i) \left(1 + \cos \left(\frac{T_{cur}}{T_i} \pi \right) \right)$$

Where where η_{min}^i and η_{max}^i are ranges for the learning rate, and T_{cur} account for how many epochs have been performed since the last restart.

Text Source: [Jason Brownlee](#)



Previously on CENG501

Global Methods: learning rate & the batch size

- Bigger batch size, bigger learning rate
- Increase batch size => increase learning rate
 - If you increase batch size from N to kN , learning rate should be scaled by:
 - \sqrt{k} [1]
 - k [2]
- Two interpretations:
 - Bigger batch means more stable gradient => Safer to make large steps.
 - Bigger batch means less number of update steps => increase learning rate to compensate.

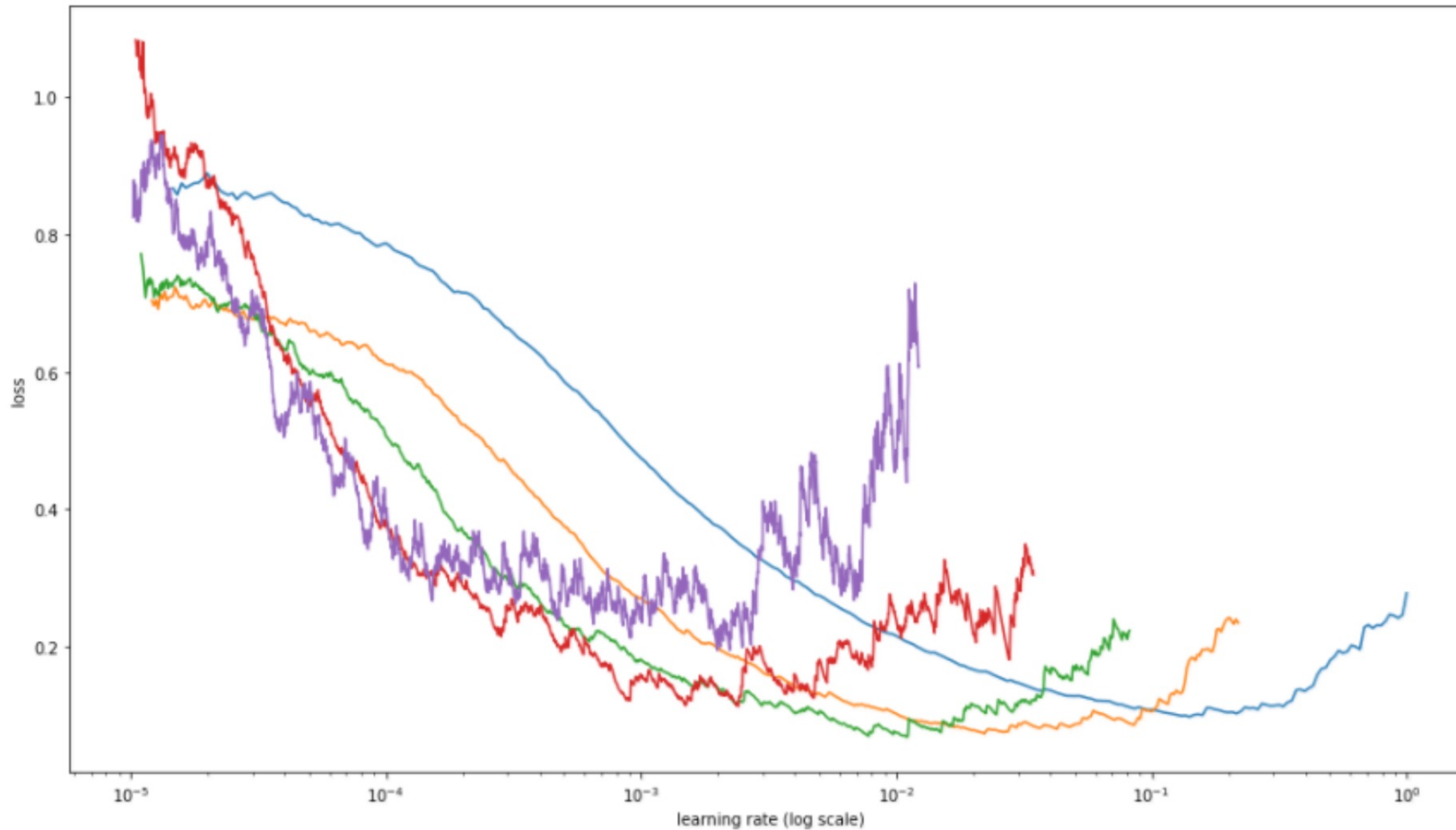
[1] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997.

[2] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677.

Global Methods: learning rate & the batch size

Previously on CENG501

BATCH SIZE:



<https://miguel-data-sc.github.io/2017-11-05-first/>

Today

- Setting the Learning Rate
- 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/2wB7ELE1BFVU4jJv7>
 - Deadline tonight

Per-parameter Methods: Adagrad

- Higher the gradient, lower the learning rate
- Accumulate square of gradients **elementwise** (initially $r = 0$):
$$r_t \leftarrow r_{t-1} + \left(\sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t) \right)^2$$
- Update each parameter/weight based on the gradient on that:

$$\Delta \theta_{t+1} \leftarrow -\frac{\eta}{\sqrt{r_t}} \sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t)$$

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter θ

Require: Small constant δ , perhaps 10^{-7} , for numerical stability

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$.

Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$. (Division and square root applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta \theta$.

end while

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7).

Algorithm taken from: Goodfellow et al., *Deep Learning*, 2016.

Per-parameter Methods: Root-Mean-Squared Propagation (RMSprop)

- Similar to Adagrad. Adagrad uses the whole history of gradients, which can be a limitation when training converges to a nice “basin”.

- RMSprop handles this by weighted/moving averaging (again, **elementwise**):

$$r_t \leftarrow \rho r_{t-1} + (1 - \rho) \left(\sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t) \right)^2$$

- ρ is typically one of: 0.9, 0.99, 0.999.
- Update each parameter/weight based on the gradient on that:

$$\Delta \theta_{t+1} \leftarrow -\frac{\eta}{\sqrt{r_t}} \sum_{i=1:M} \nabla_{\theta_t} \mathcal{L}(\mathbf{x}_i; \theta_t)$$

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ , decay rate ρ

Require: Initial parameter θ

Require: Small constant δ , usually 10^{-6} , used to stabilize division by small numbers

Initialize accumulation variables $\mathbf{r} = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$.

 Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.

 Compute parameter update: $\Delta \theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{\delta + \mathbf{r}}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$.

end while

Currently, unpublished. Proposed by Hinton in one of his lectures.

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.

Per-parameter Methods: RMSprop with Nesterov Momentum

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ , decay rate ρ , momentum coefficient α

Require: Initial parameter θ , initial velocity v

Initialize accumulation variable $r = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

Compute interim update: $\tilde{\theta} \leftarrow \theta + \alpha v$.

Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$.

Accumulate gradient: $r \leftarrow \rho r + (1 - \rho) \mathbf{g} \odot \mathbf{g}$.

Compute velocity update: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot \mathbf{g}$. ($\frac{1}{\sqrt{r}}$ applied element-wise)

Apply update: $\theta \leftarrow \theta + v$.

end while

Algorithm taken from: Goodfellow et al., Deep Learning, 2016.

Per-parameter Methods: Adaptive Moments (Adam)

- A variation of RMSprop + momentum
- Incorporates first & second order moments
- Bias correction needed to get rid of bias towards zero at initialization

Algorithm taken from:
Goodfellow et al., Deep Learning, 2016.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$.
(Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters θ

Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

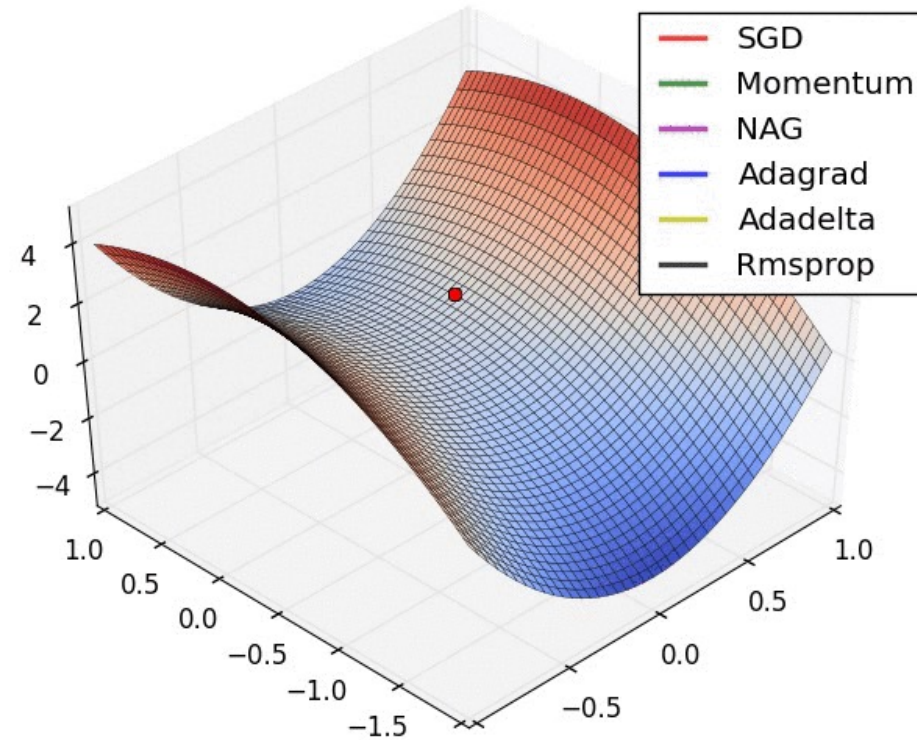
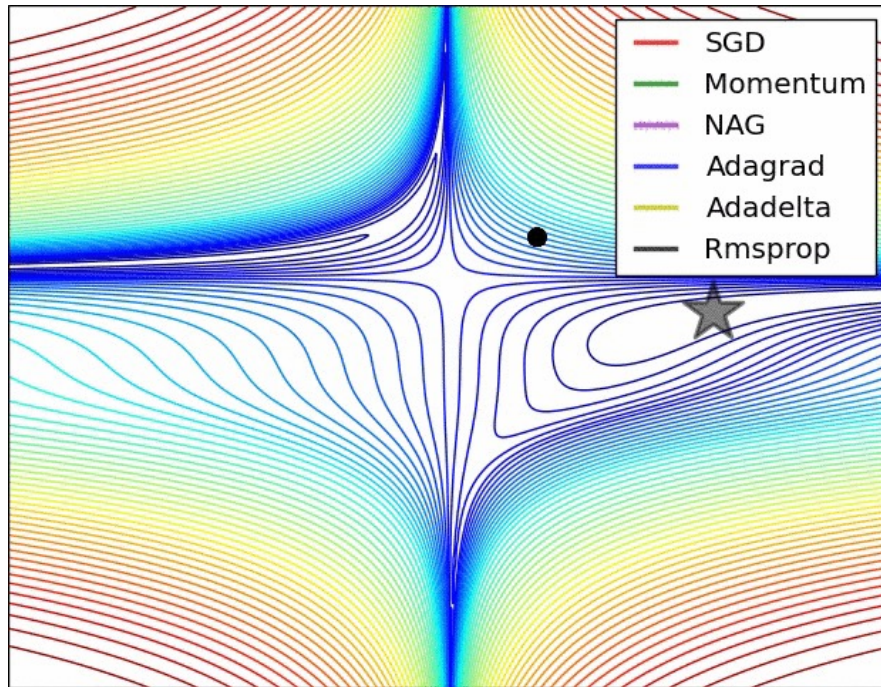
Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

Compute update: $\Delta \theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$ (operations applied element-wise)

Apply update: $\theta \leftarrow \theta + \Delta \theta$

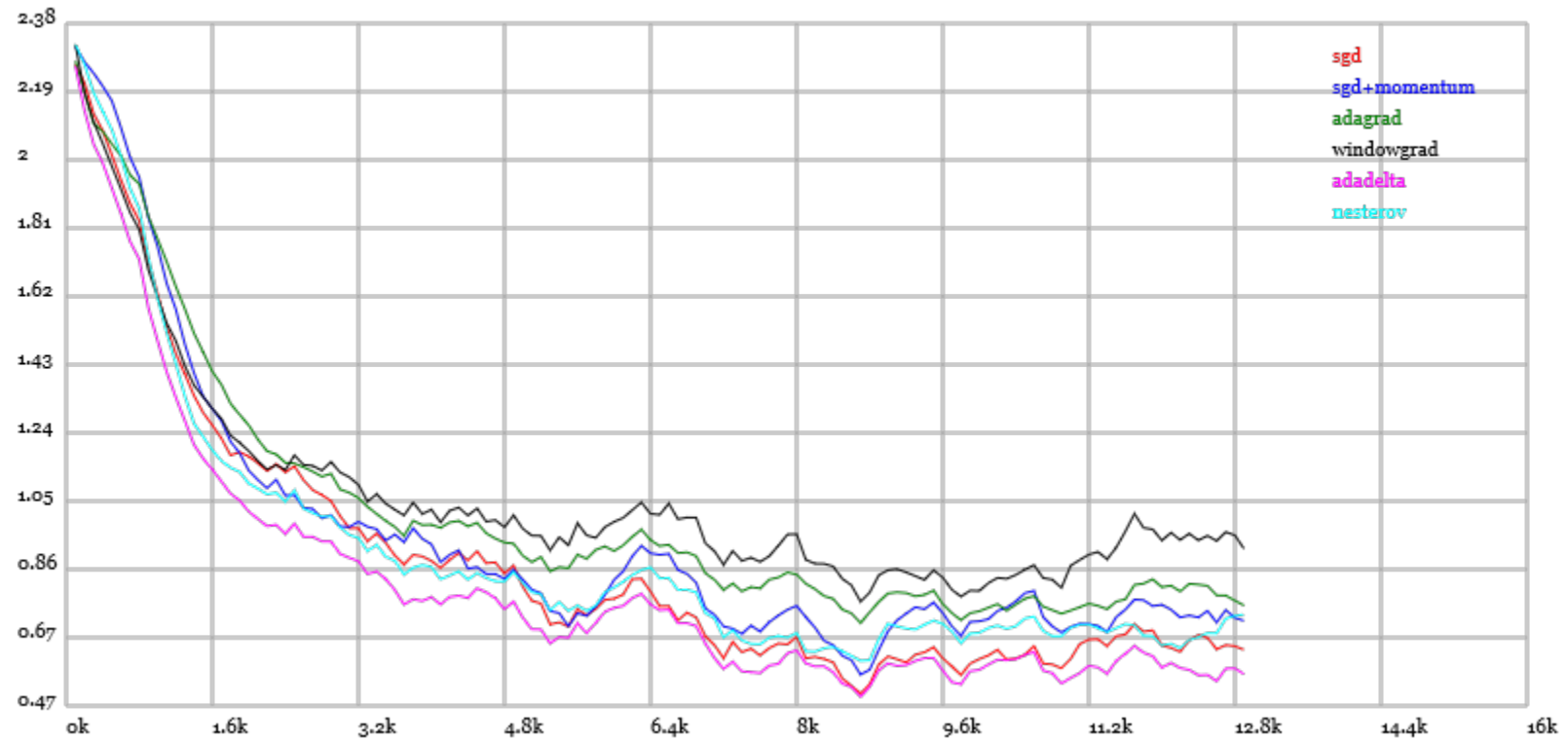
end while

Comparison



NAG: Nesterov's Accelerated Gradient

Comparison



- When SGD+momentum is tuned for hyperparameters, it can outperform Adam etc.
- There are methods that try to finetune the hyper-parameters:

YellowFin and the Art of Momentum Tuning

<https://arxiv.org/abs/1706.03471>

To sum up

- Different problems seem to favor different per-parameter methods
- Adam seems to perform better among per-parameter adaptive learning rate algorithms
- SGD+Nesterov momentum seems to be a fair alternative

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), 251-257.

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

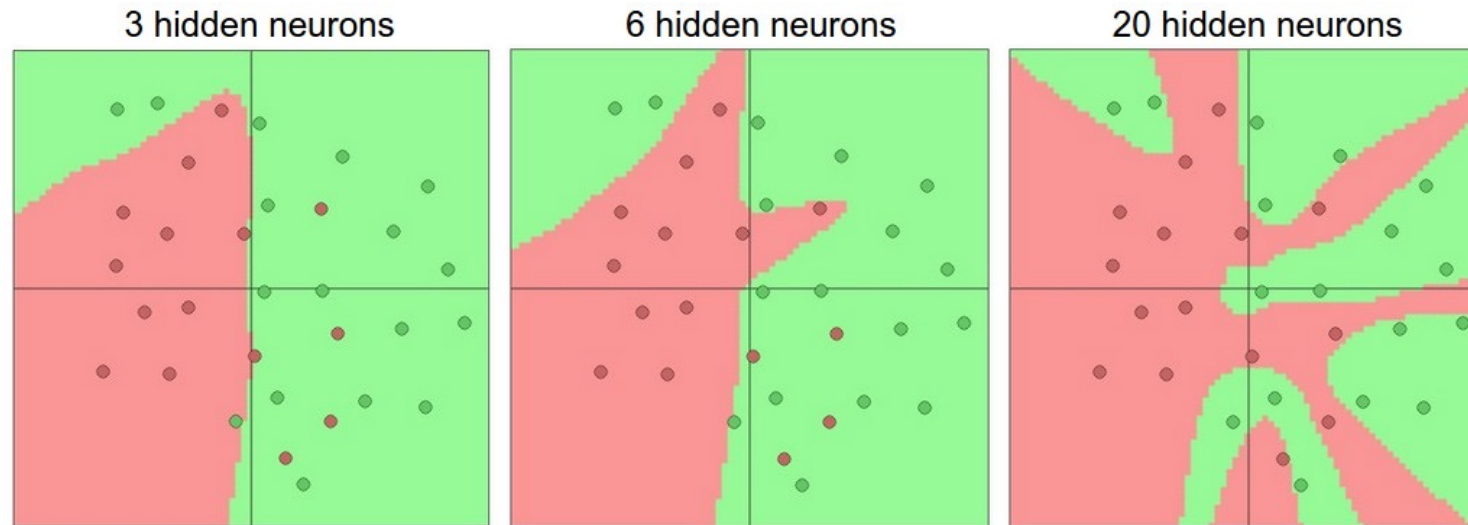


Figure: <https://cs231n.github.io/>

- 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)

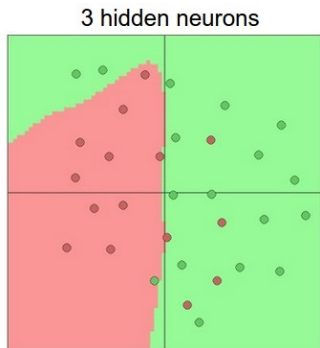
- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be:
 - $\frac{2}{3} \times$ (the size of the input layer + the size of the output layer)
- The number of hidden neurons should be less than twice the size of the input layer.

Number of hidden layers

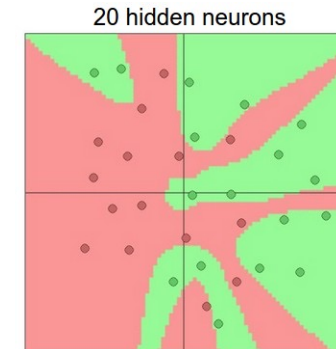
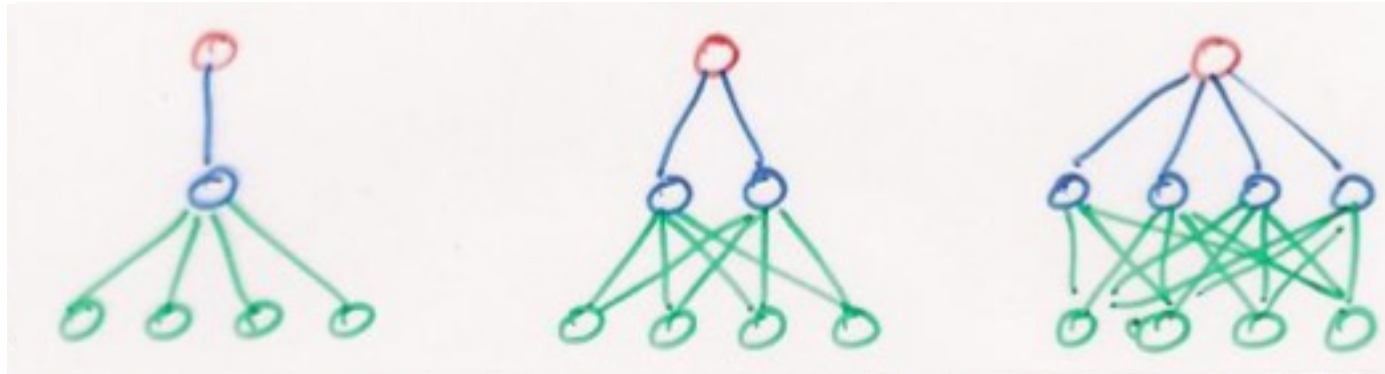
- Depends on the nature of the problem
 - Linear classification? → No hidden layers needed
 - Non-linear classification?

Model Complexity

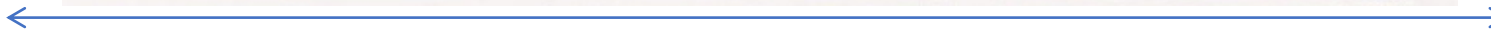
- Models range in their flexibility to fit arbitrary data



<https://cs231n.github.io/>



<https://cs231n.github.io/>



highly constrained

lowly constrained

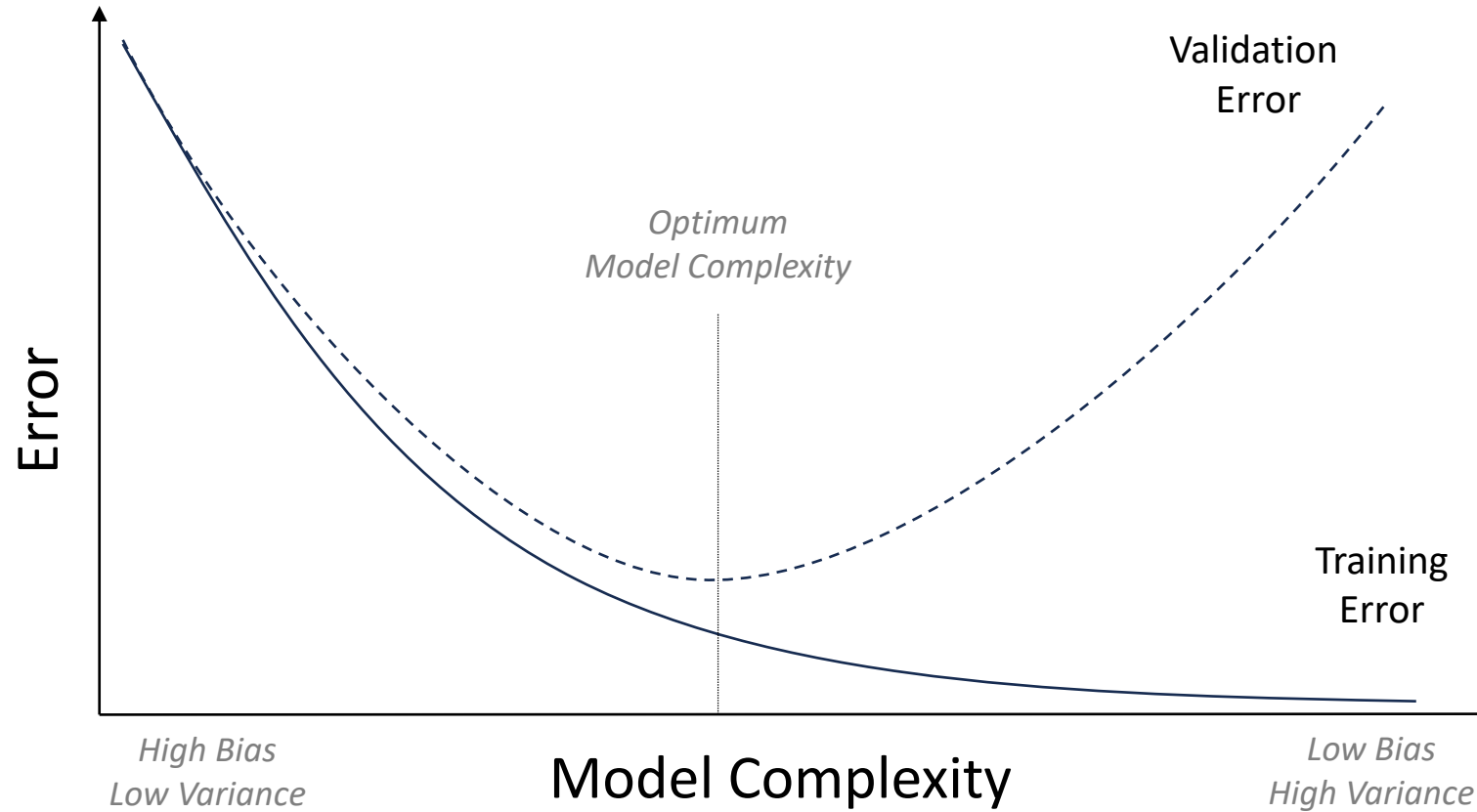
low variance

high variance

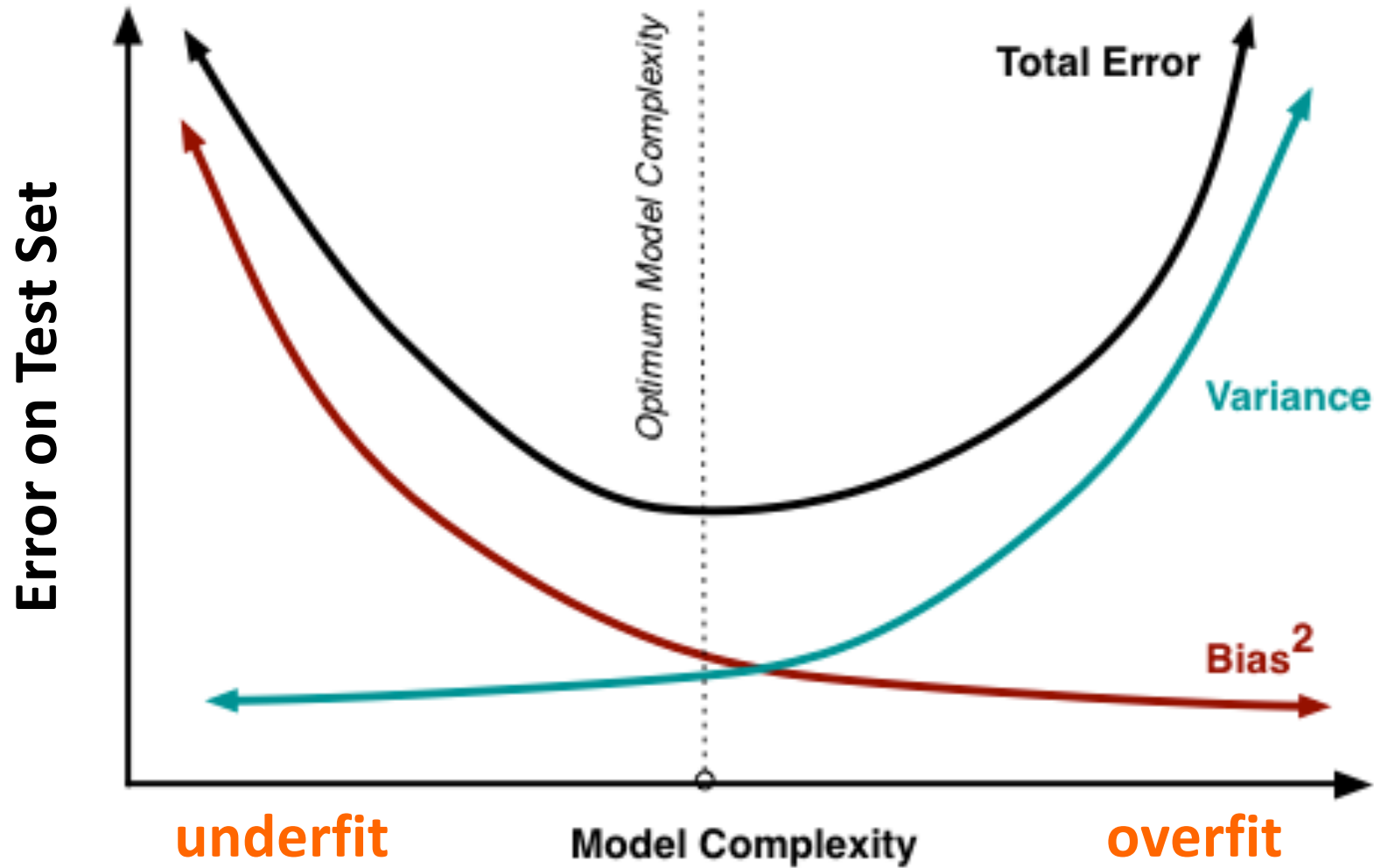
small capacity may prevent it from representing all structure in data

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

Training Vs. Test/Val Set Error



Bias-Variance Trade Off



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 $\text{err}_P(h)$ relative

Double Descent

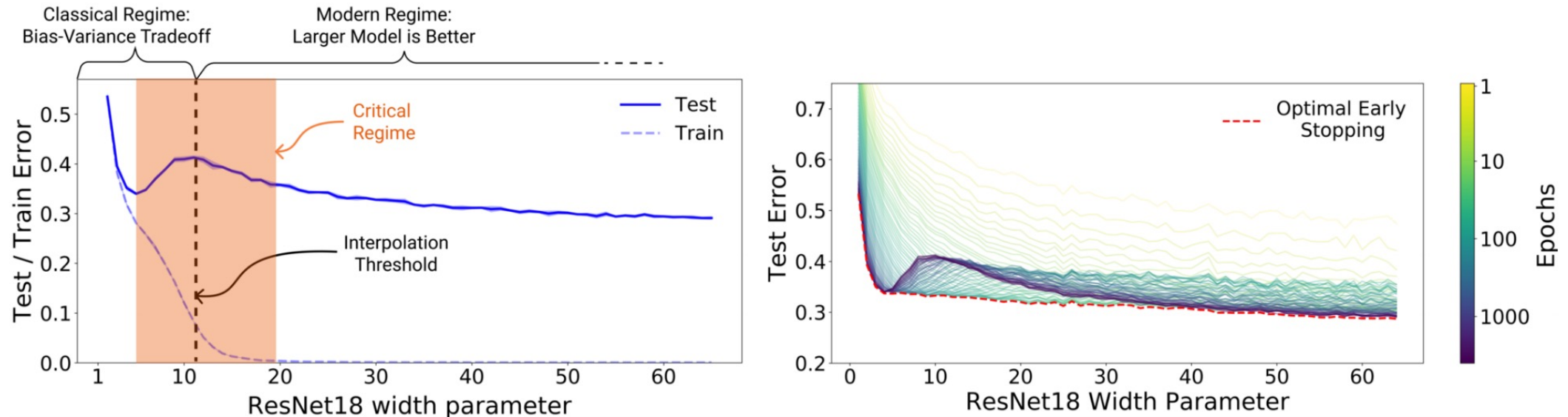
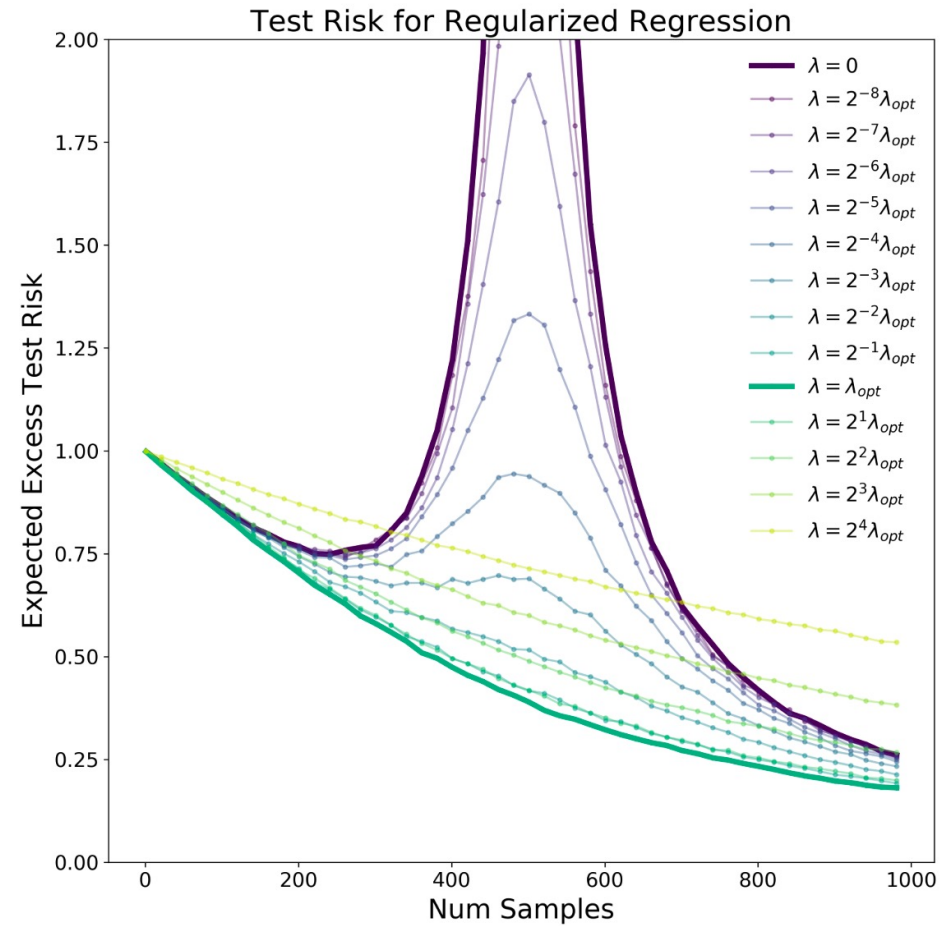


Figure 1: **Left:** Train and test error as a function of model size, for ResNet18s of varying width on CIFAR-10 with 15% label noise. **Right:** Test error, shown for varying train epochs. All models trained using Adam for 4K epochs. The largest model (width 64) corresponds to standard ResNet18.

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

Double Descent



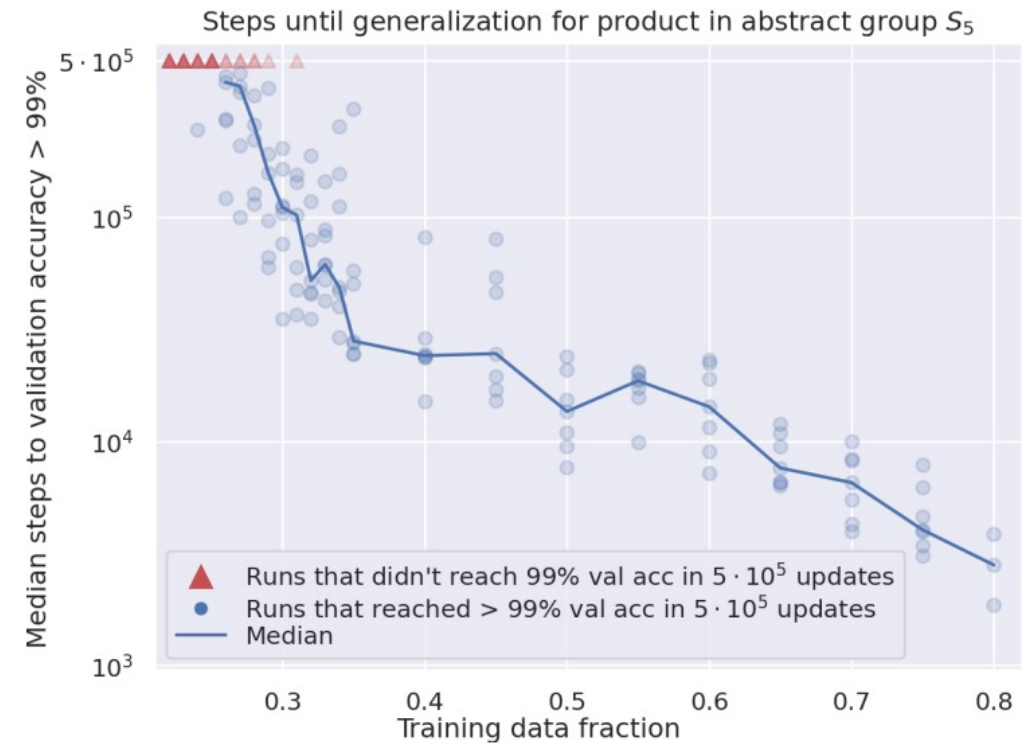
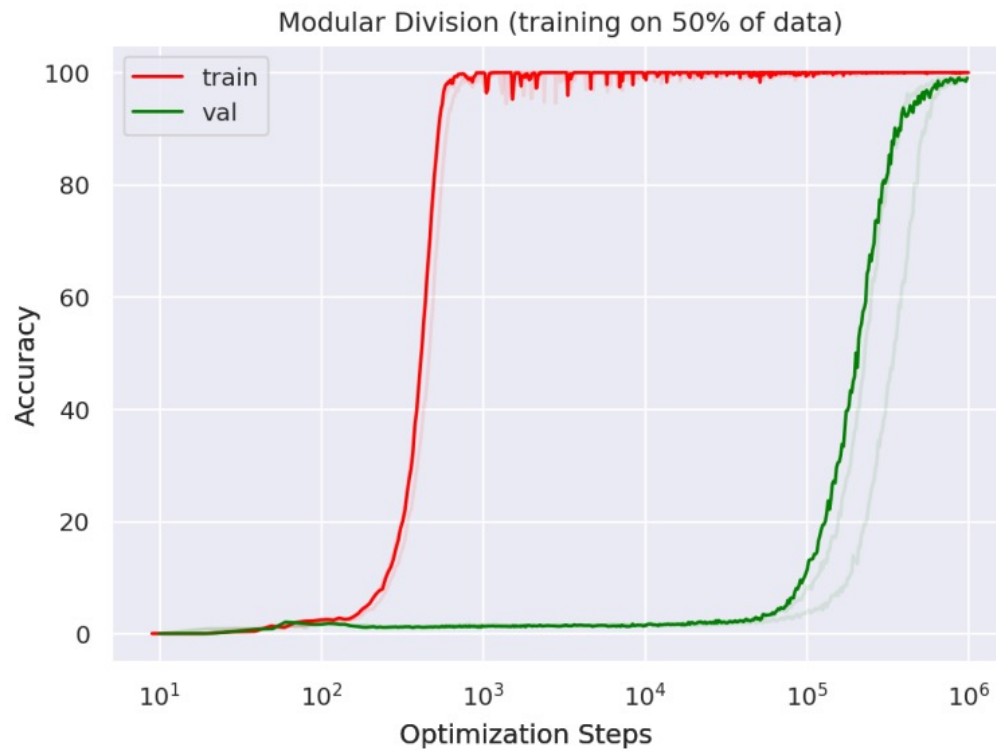
Nakkiran et al., “Optimal Regularization Can Mitigate Double Descent”, 2020.

Grokking

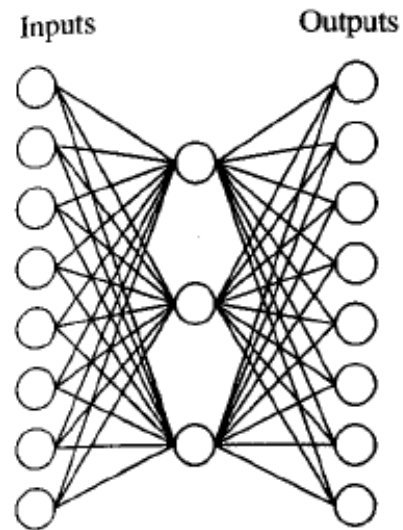
[LG] 6 Jan 2022

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?



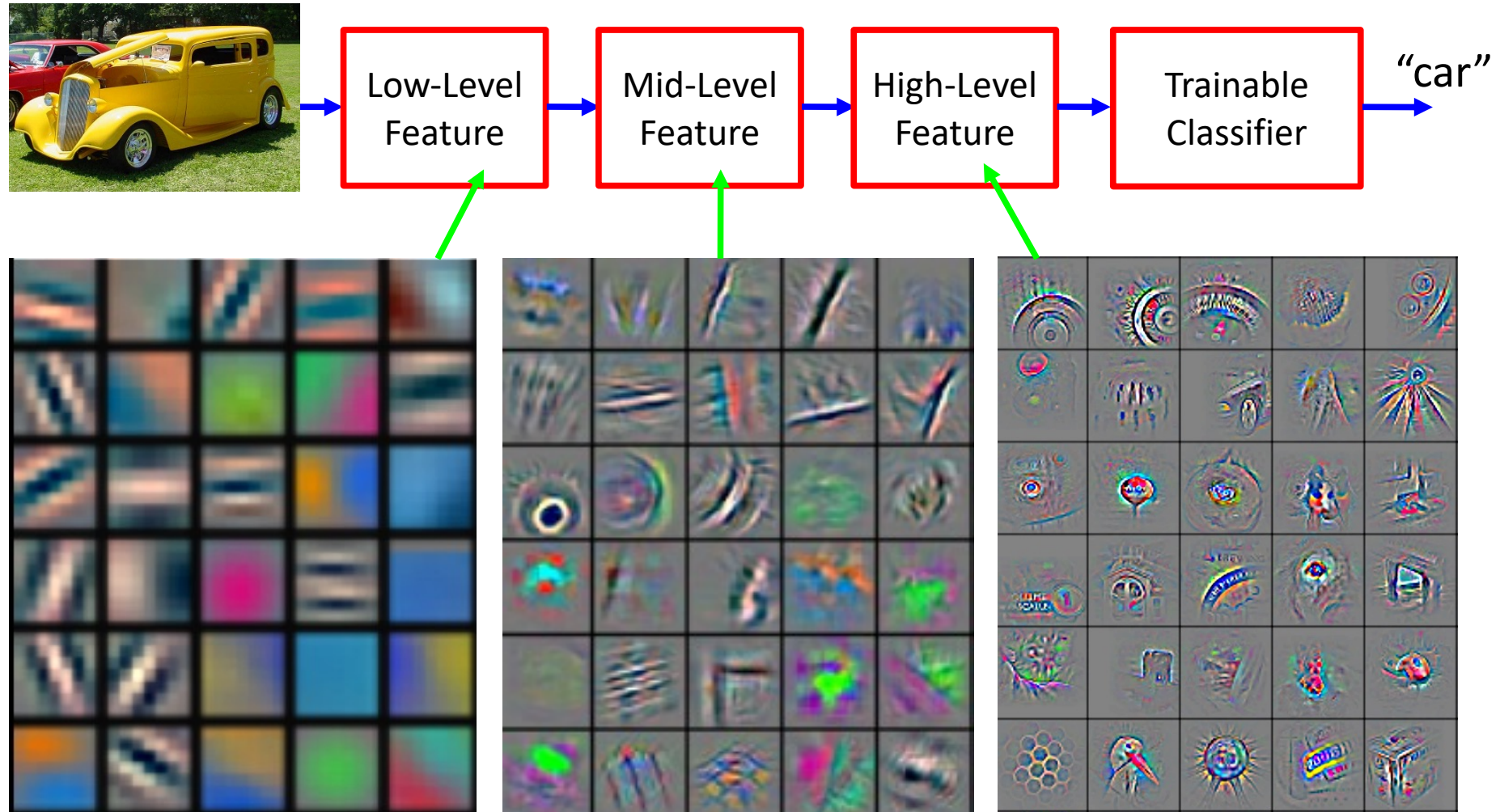
| Input | | Hidden Values | | | | Output |
|----------|---|---------------|-----|-----|---|----------|
| 10000000 | → | .89 | .04 | .08 | → | 10000000 |
| 01000000 | → | .15 | .99 | .99 | → | 01000000 |
| 00100000 | → | .01 | .97 | .27 | → | 00100000 |
| 00010000 | → | .99 | .97 | .71 | → | 00010000 |
| 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.

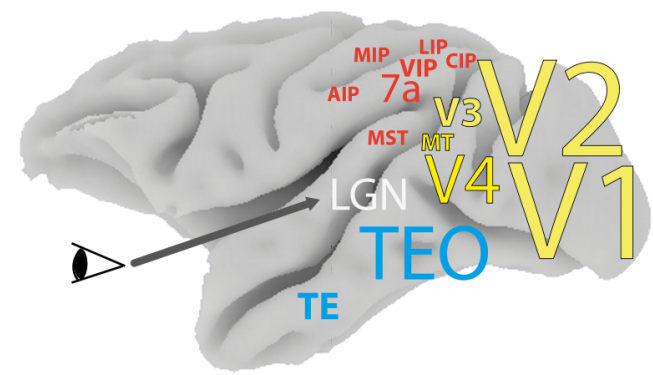
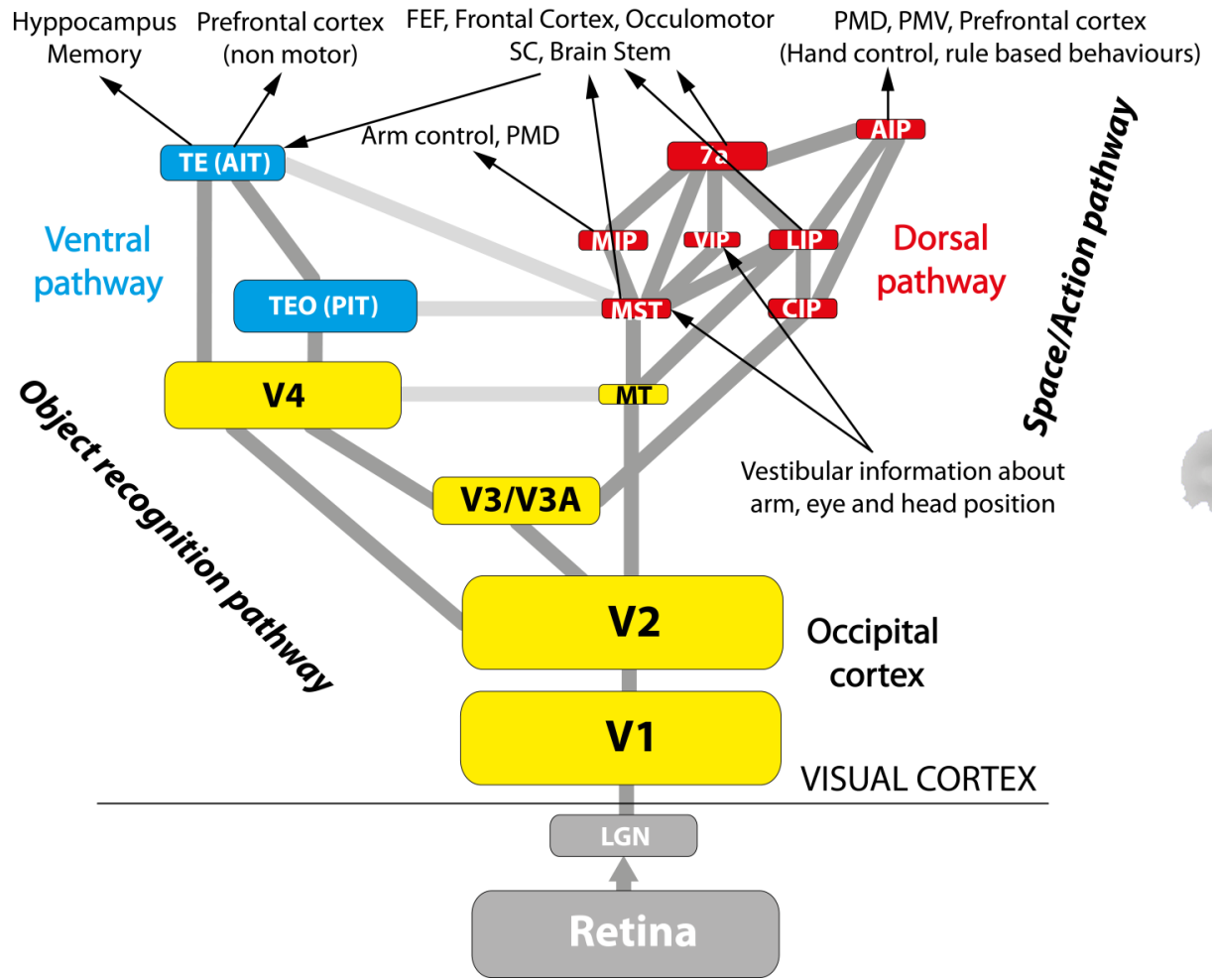
T. Mitchell, "Machine Learning", 1997.

What do the layers represent?



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

Similarities to the Hierarchies in Visual Cortex

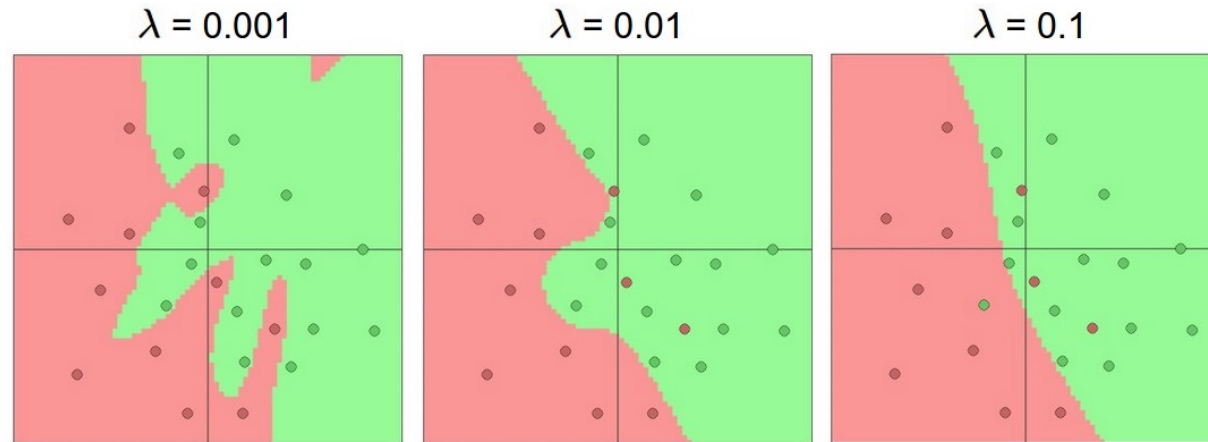


Krueger, Janssen, Kalkan, Lappe, .., "Deep Hierarchies in the Primate Visual Cortex: What Can We Learn For Computer Vision", IEEE PAMI, 2013.

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

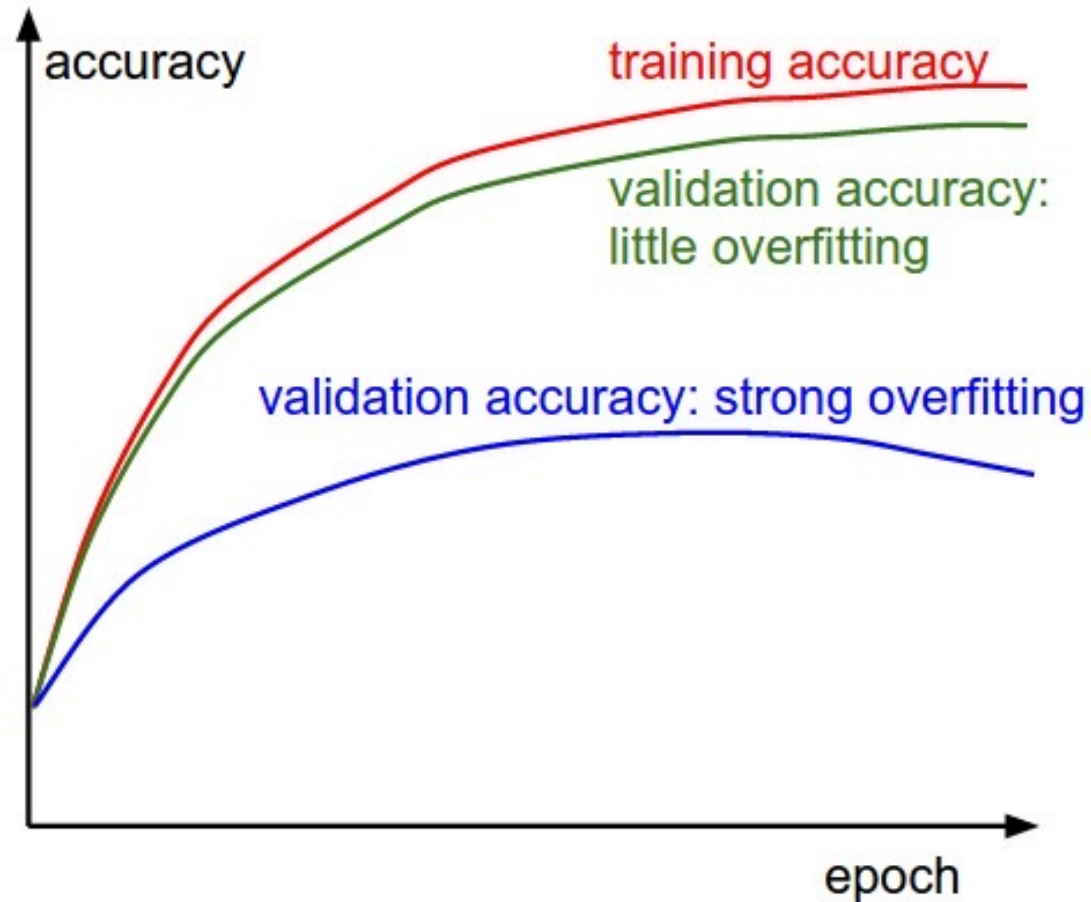


CENG501

Adapted from Michael Mozer

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

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**

Incorporating Domain-Appropriate Bias Into Model

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

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

Adding bias into a model


- Adding hidden layers or direct connections based on the problem

Direct I/O connections to learn easy parts of task


- Nettalk performs at about 70% without hidden units
(guess)



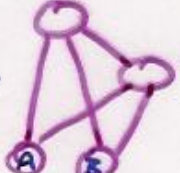
Performance up to 100% with hidden units



- Hidden units useful for handling exceptions
- E.g., XOR



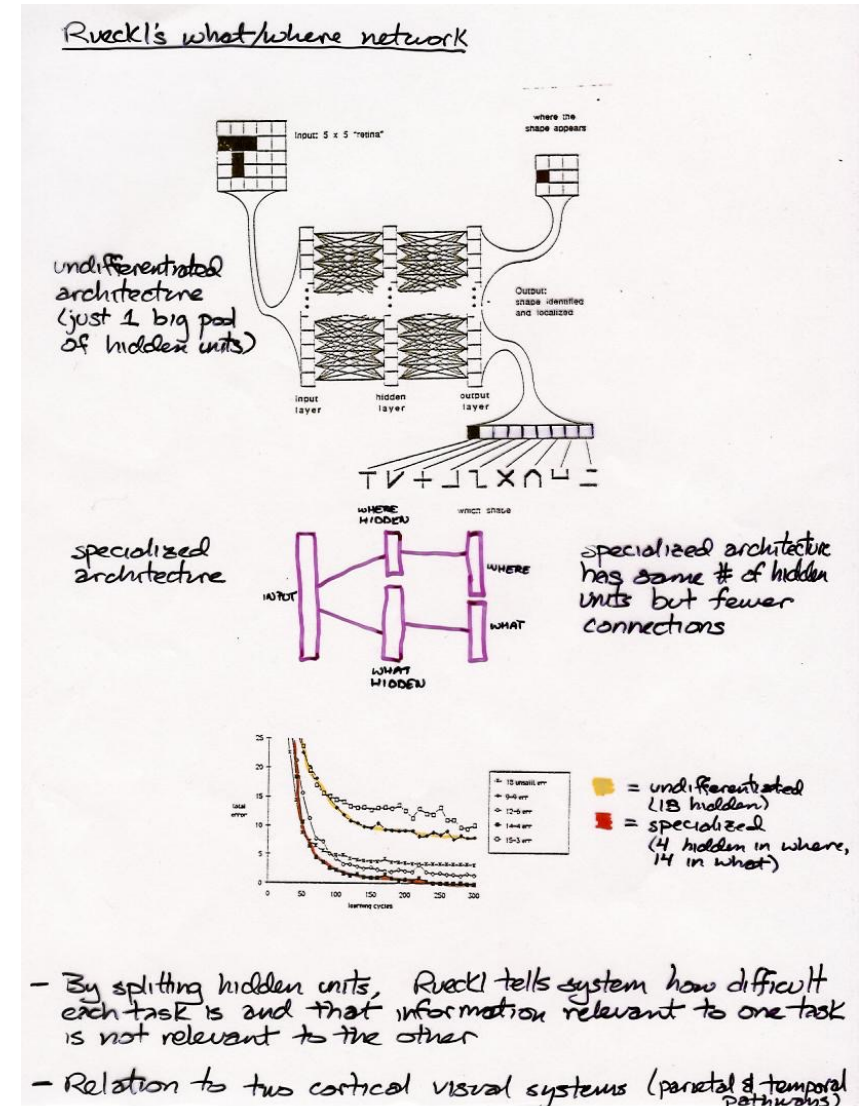
for easy parts of task



hidden units discover higher order features critical to performance (here, A & B)

Adding bias into a model


- Modular architectures
 - Specialized hidden units for special problems




Adding bias into a model

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

Constraints on activities
e.g. reduce amount of information flowing through net by encouraging binary-valued hidden units

$$E = \sum_p \sum_i (d_i^p - o_i^p)^2 + \sum_{n \in \text{hidden}} o_n(1 - o_n)$$


Constraints among weights
E.g., T-C problem: Each hidden unit should detect the same feature, but shifted in position



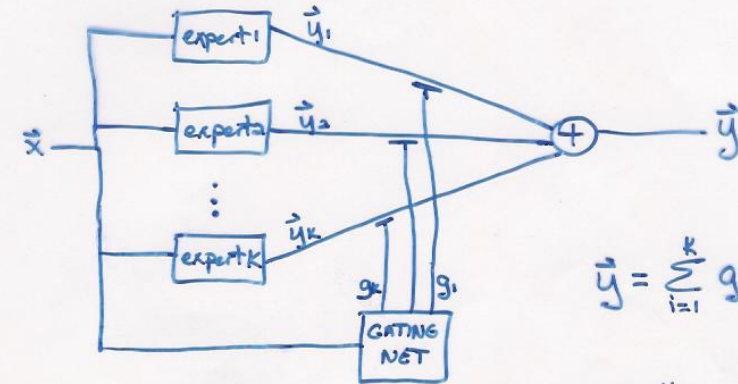
Set $w_1 = w_2$ initially
 $\Delta w_1 = \Delta w_2 = -\epsilon (\partial \epsilon / \partial w_1 + \partial \epsilon / \partial w_2)$

Adding bias into a model

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

Specialized Activation Functions

mixture of experts (Jacobs, Jordan, Nowlan, & Hinton 91)



$$\hat{y} = \sum_{i=1}^k g_i \cdot \hat{y}_i$$

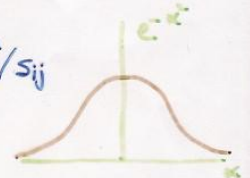
$$E = -\ln \sum_{i=1}^k g_i e^{-\frac{1}{2} \|d - g_i\|^2}$$

Radial Basis Functions

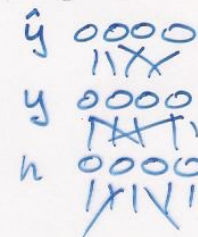


$$net_i = \sum (x_j - w_{ij})^2 / s_{ij}$$

$$y_i = e^{-net_i}$$



Normalized Exponential Transform, a.k.a. softmax (Bridle 91)



$$y_i = \sum w_{ij} h_j \quad (\text{linear})$$

$$\hat{y}_i = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

for classification:
 $E = -\ln \hat{y}_d$
index of desired output

NOTE: $0 \leq \hat{y}_i \leq 1$
 $\sum \hat{y}_i = 1$

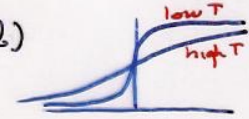
Adding bias into a model

- 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)



$$O_i = \frac{1}{1 + e^{-\text{net}_i/T_i}}$$

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

- E.g., input saliency term

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

$$\text{net}_i = \sum_{\text{layer } a} W_{ij} O_j S_j$$



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

Equivalent to changing all outgoing weights from input unit simultaneously

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

Regularization

- Regularization strength can effect overfitting

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

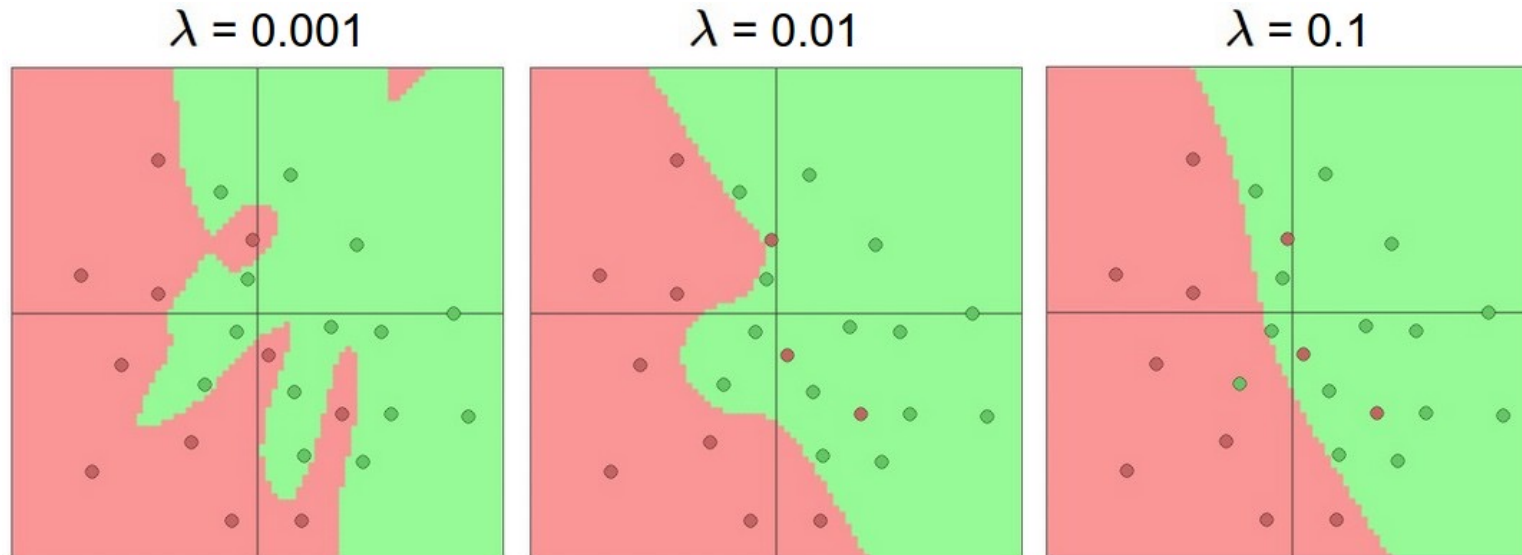


Figure: <https://cs231n.github.io/>

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

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



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

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

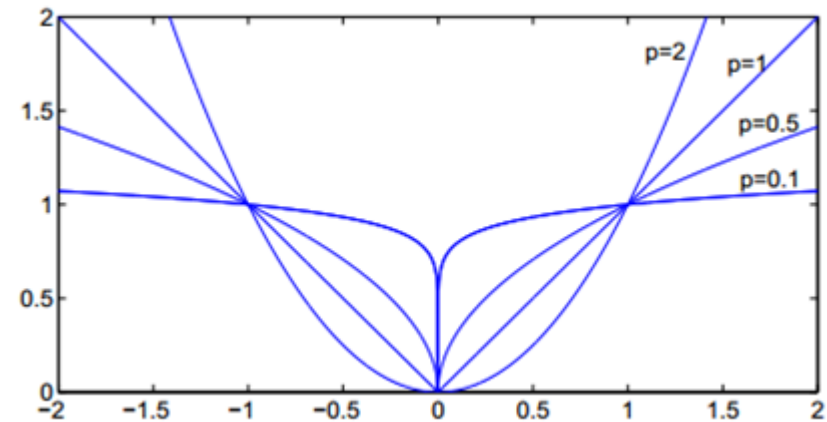
Weight Decay

- 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>

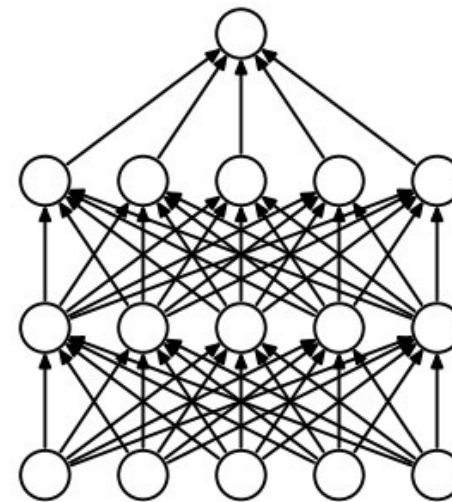
<https://arxiv.org/abs/1911.05920>

Large-norm L2 regularization

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

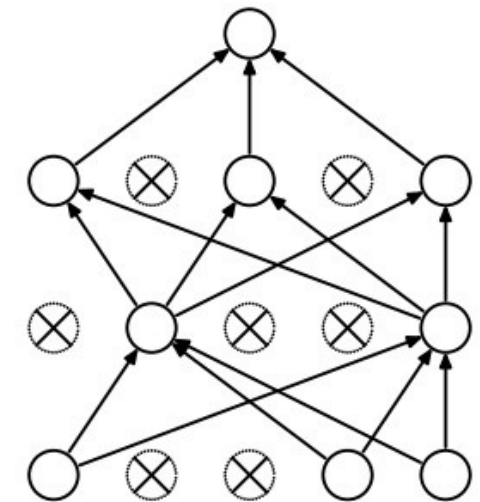
Regularization

- Enforce an upper bound on weights:
 - Max norm:
 - $\|w\|_2 < c$
 - 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!



(a) Standard Neural Net

Fig: Srivastava et al., 2014



(b) After applying dropout.

Regularization: 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 .

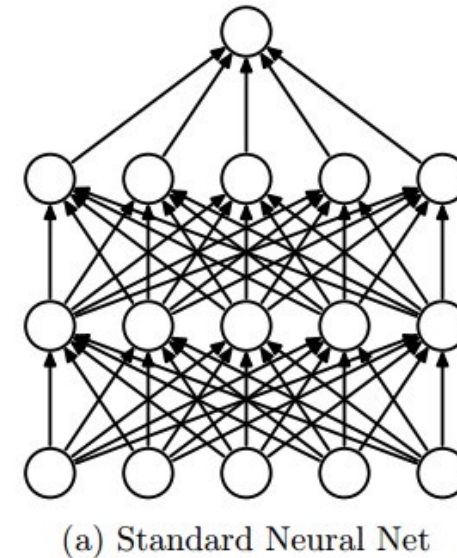
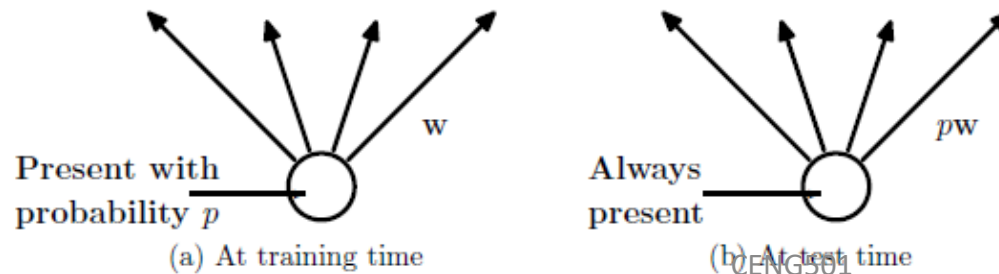


Fig: Srivastava et al., 2014

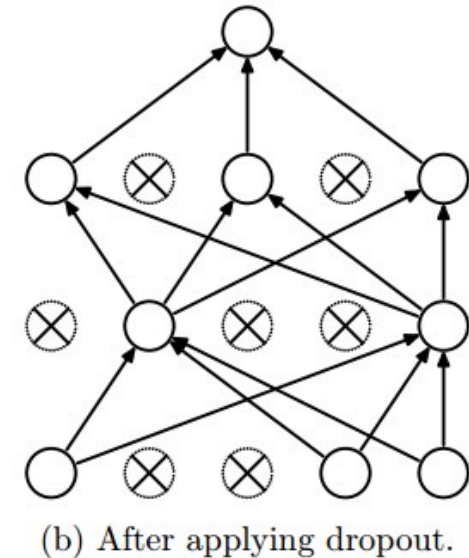


Fig: Srivastava et al., 2014

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
```

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
```

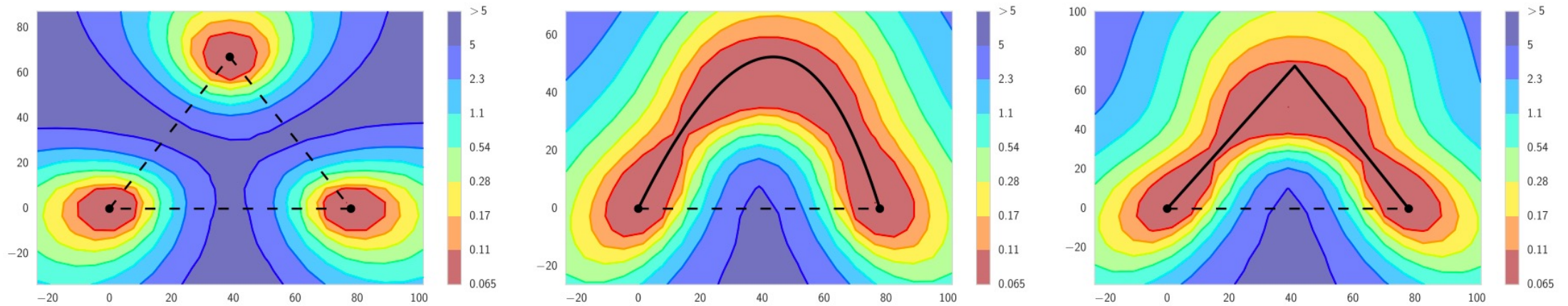
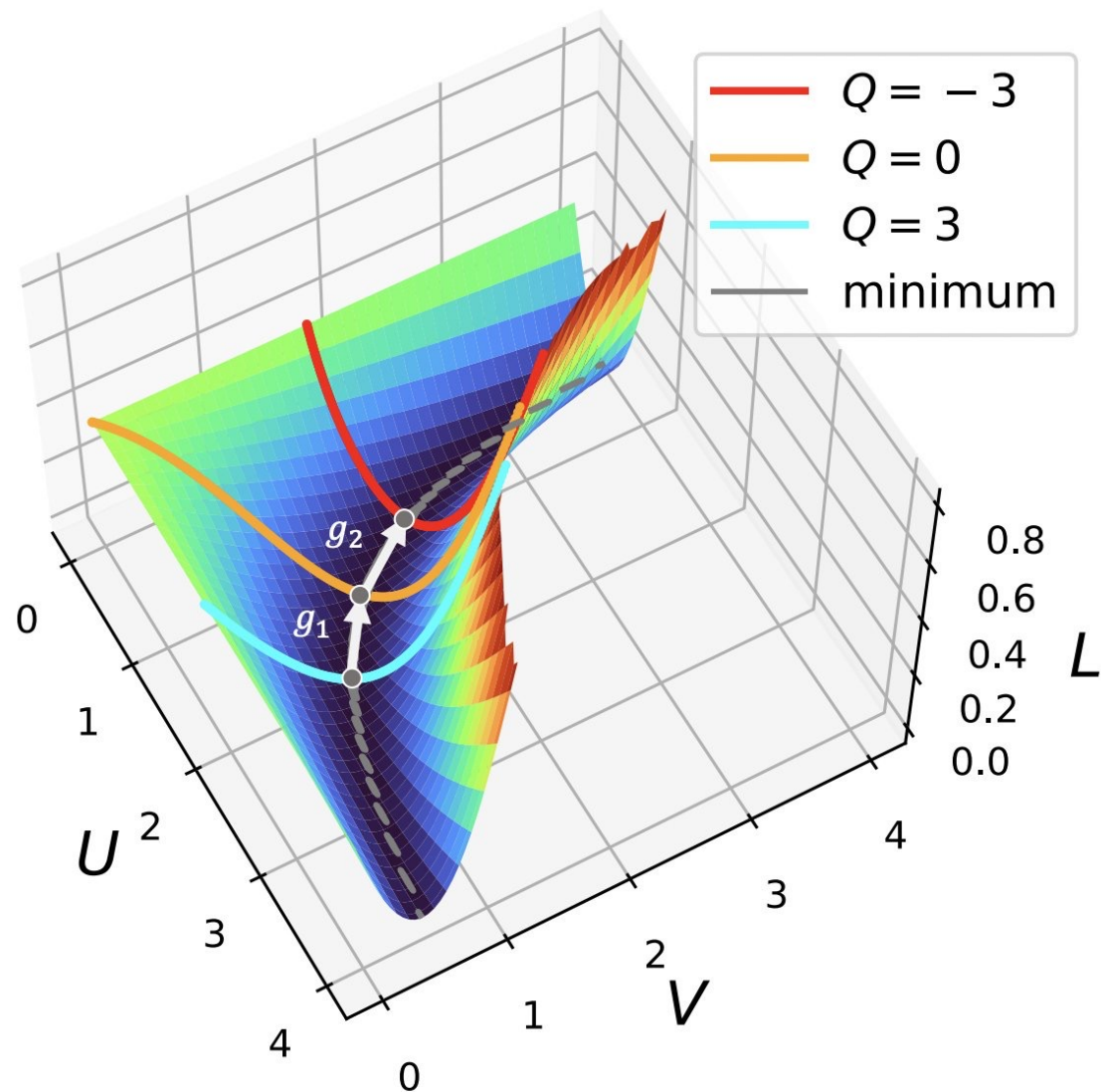


Figure 1: The ℓ_2 -regularized cross-entropy train loss surface of a ResNet-164 on CIFAR-100, as a function of network weights in a two-dimensional subspace. In each panel, the horizontal axis is fixed and is attached to the optima of two independently trained networks. The vertical axis changes between 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

Jordan 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>

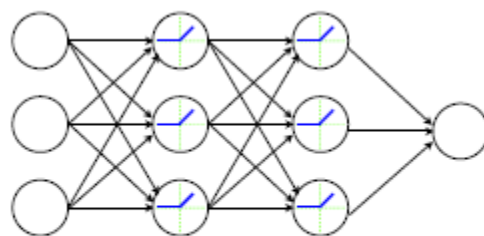
Drop-Activation: Implicit Parameter Reduction and Harmonic Regularization

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

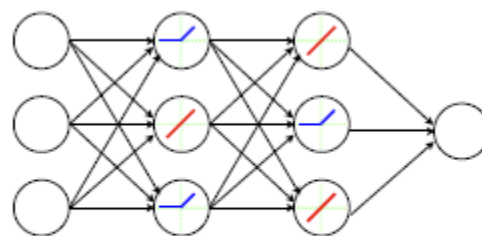
Yuehaw Kwoo
Stanford University
450 Serra Mall
Stanford, CA 94305
ykhoo@stanford.edu

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



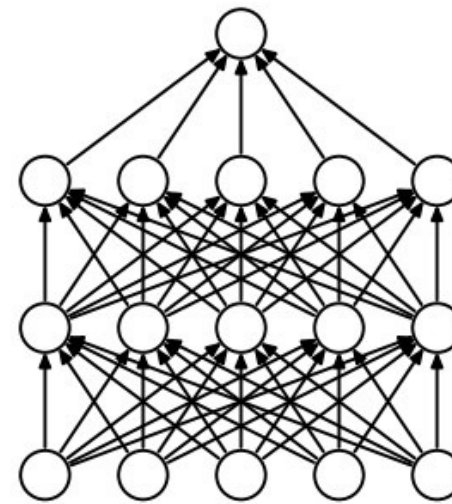
(b) After applying Drop-Activation

| 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.

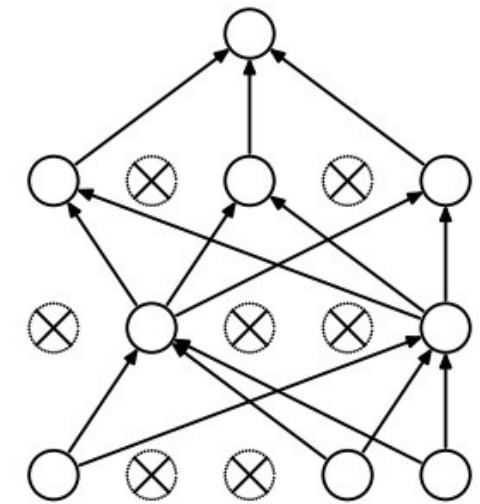
Dropout as Ensemble Training Method

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



(a) Standard Neural Net

Fig: Srivastava et al., 2014



(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

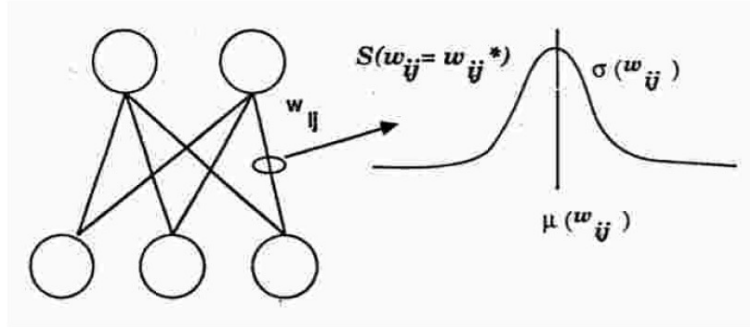


Figure 1: SDR sampling.

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

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

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

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).

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

Lottery Ticket Hypothesis

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

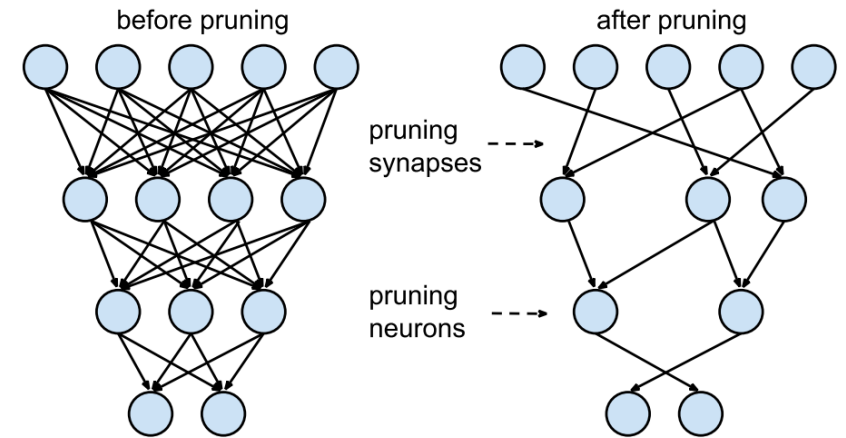


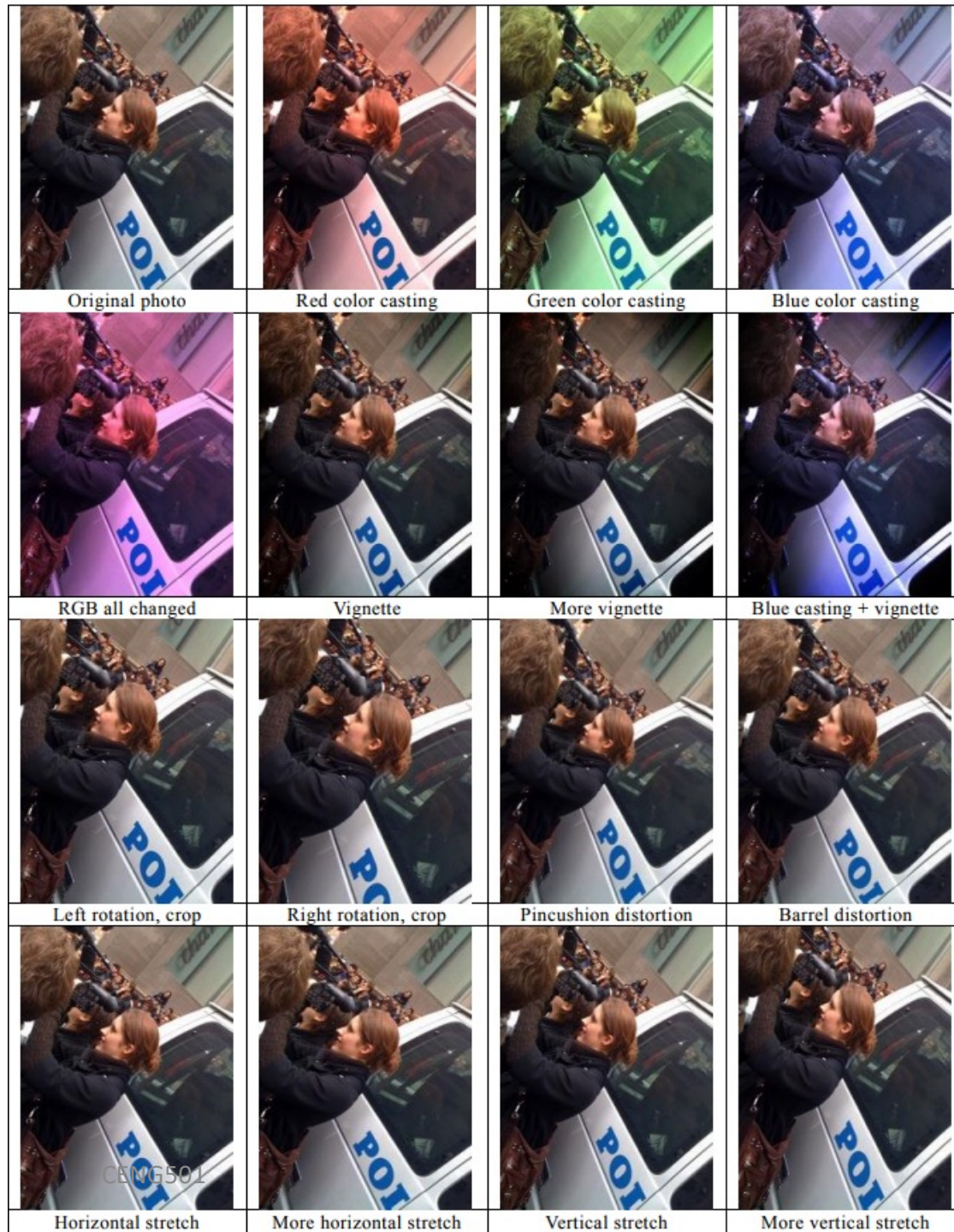
Figure: <https://herbiebradley.com/The-Lottery-Ticket-Hypothesis>

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



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)

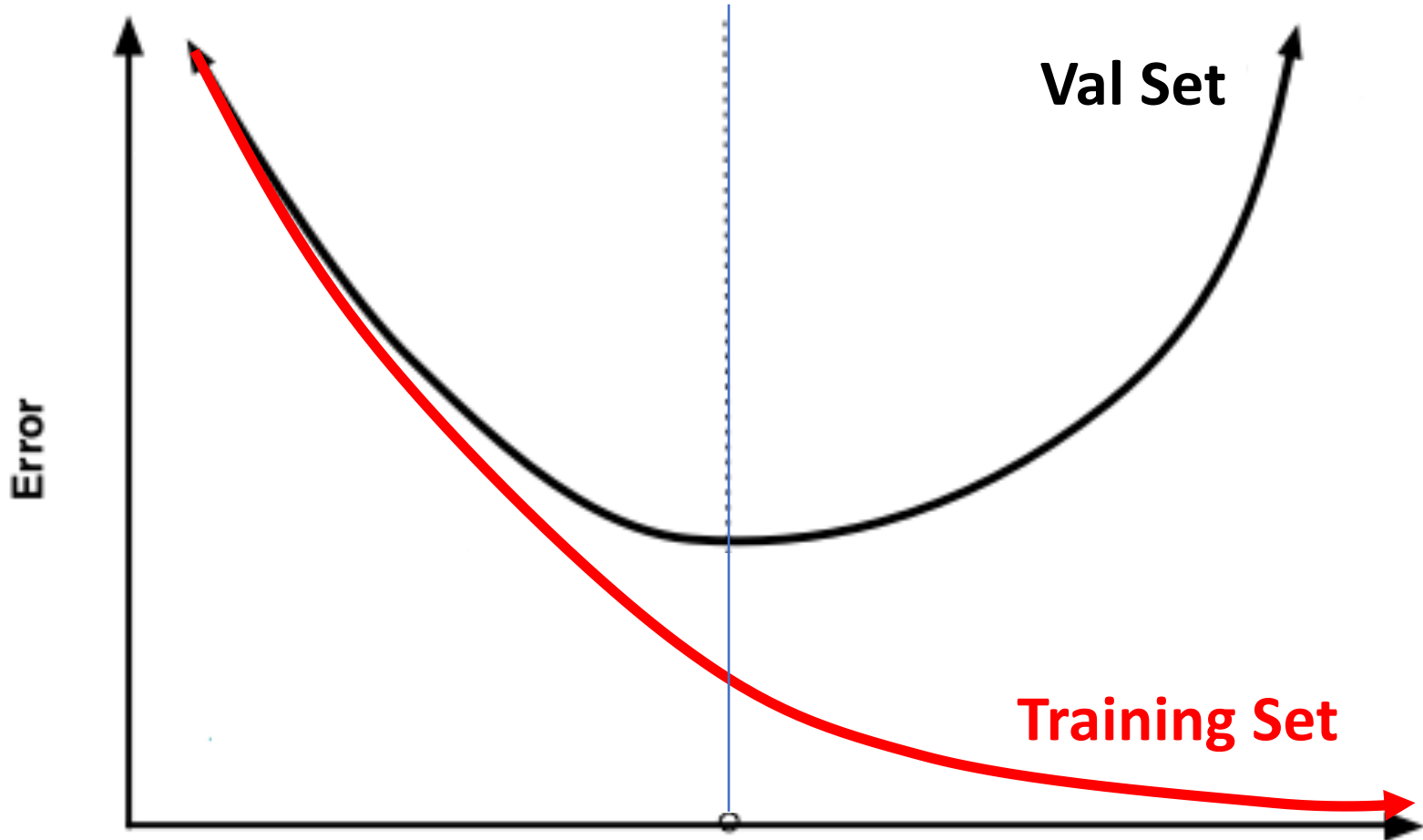
When To Stop Training

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

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

- 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

Training Vs. Val Set Error



Data Preprocessing and weight initialization

Data Preprocessing

- Mean subtraction
- Normalization
- PCA and whitening

Data Preprocessing: Mean subtraction

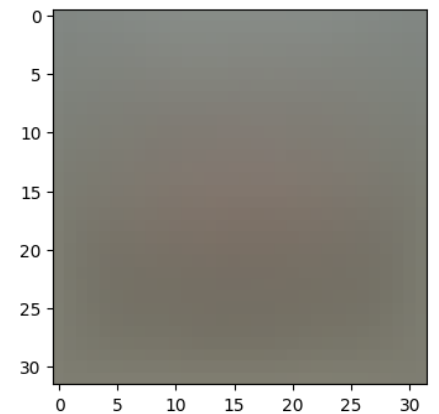
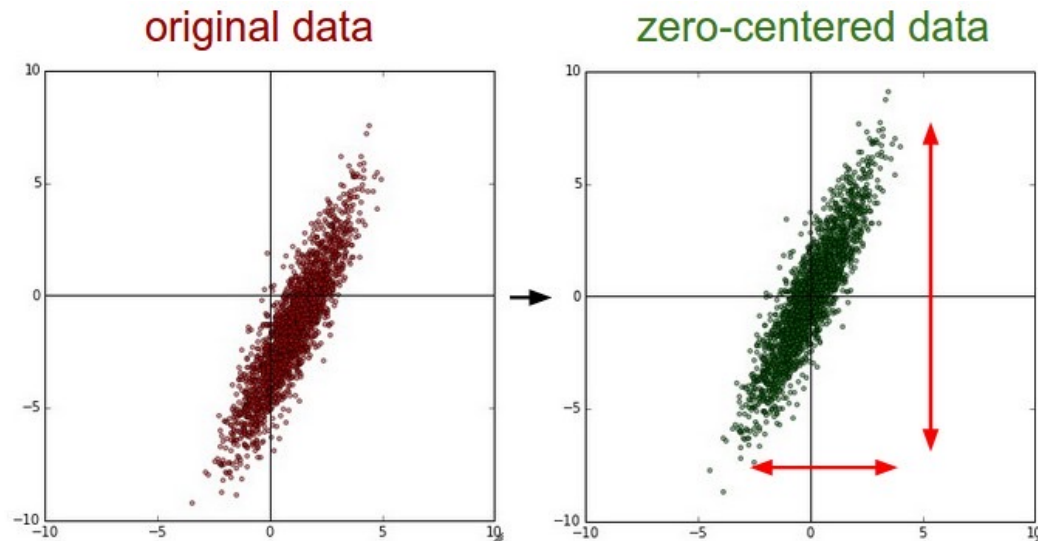
- Compute the mean of each dimension, μ_i , over the training set:

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

- Subtract the mean for each dimension:

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

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



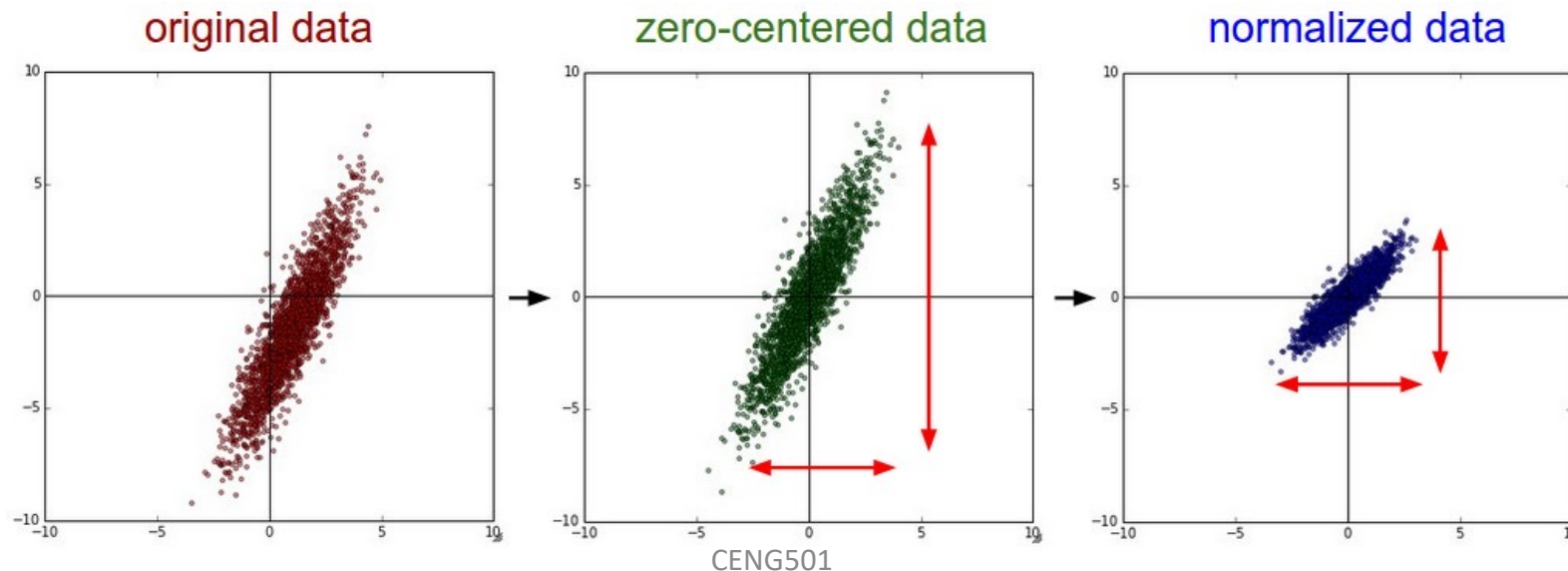
Mean image of CIFAR10
(from PA1)

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:

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

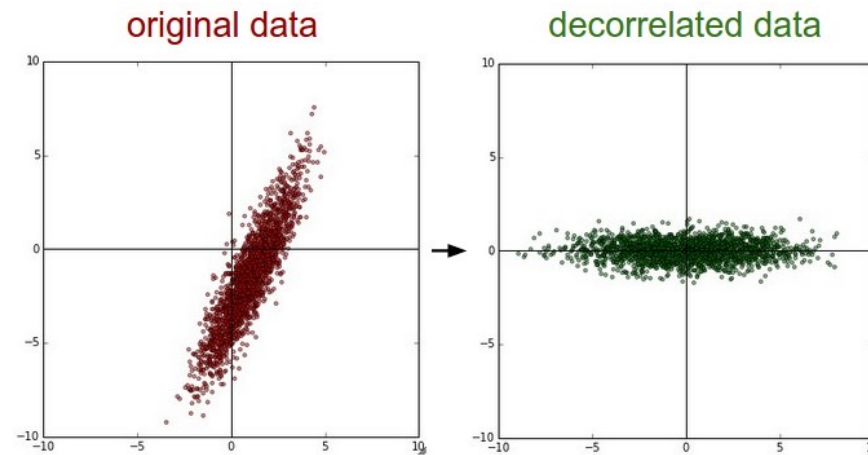
- Effect: Make the dimensions have the same scale



CENG501

Data Preprocessing: Principle Component Analysis

- First center the data
- Find the eigenvectors e_1, \dots, e_n
- Project the data onto the eigenvectors:
 - $x_i^R = x_i \cdot [e_1, \dots, e_n]$
- 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



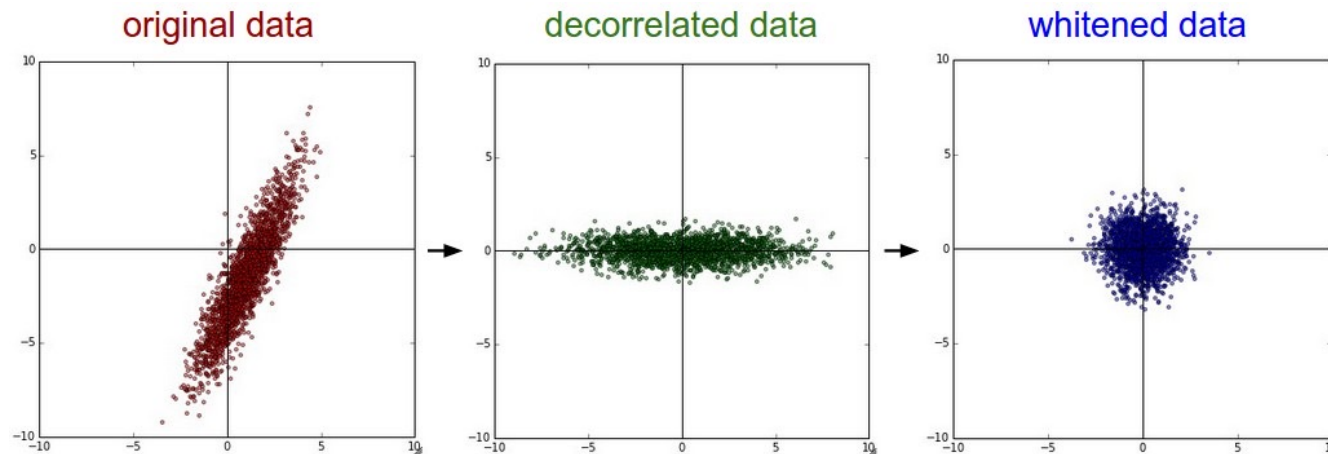
CENG501

Data Preprocessing: Whitening

- Normalize the scale with the norm of the eigenvalue:

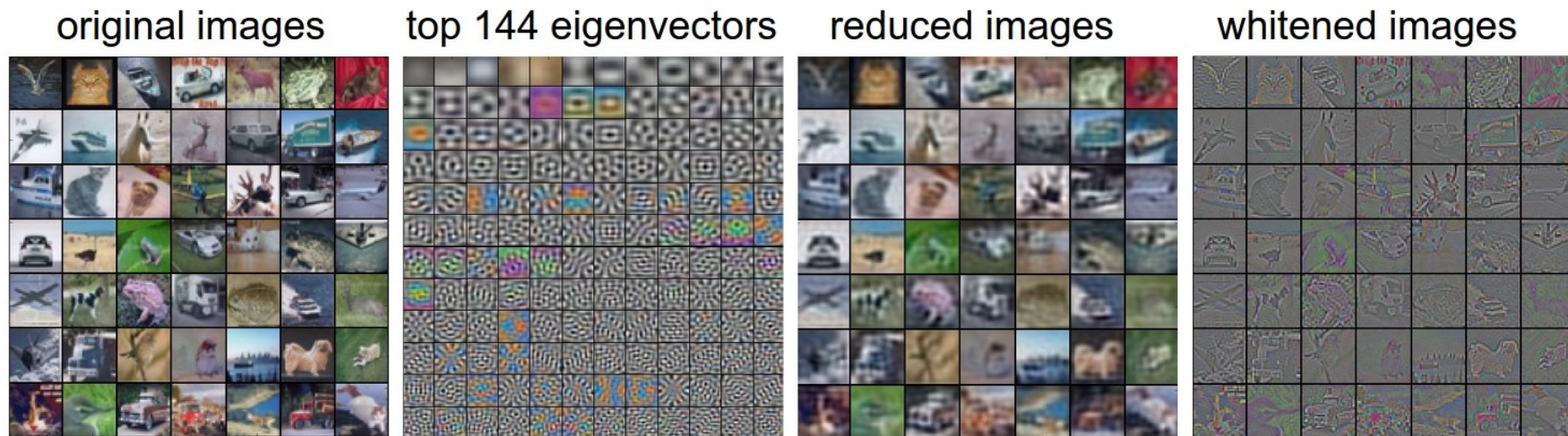
$$x_i^W = x_i^R / (\lambda_i + \epsilon)$$

- 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.



CENG501

Data Preprocessing: Example



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., $\text{Normal}(0,0.01)$
 - 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 $\text{Var}(X) = E[(X - \mu)^2]$):

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

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].

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

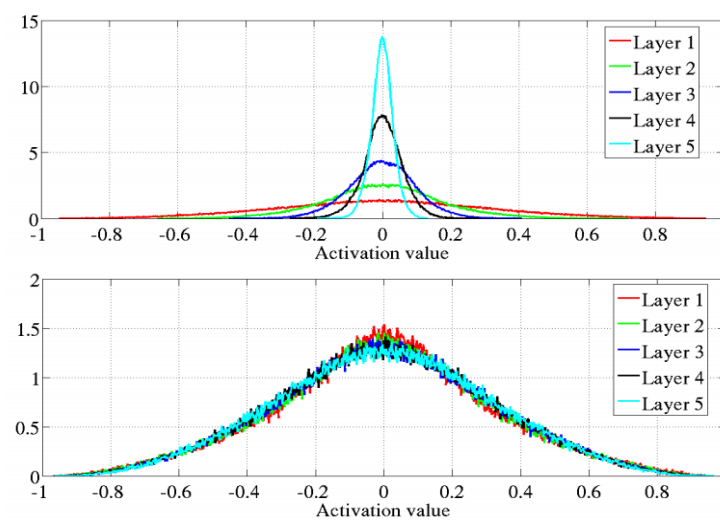


Figure 6: Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.

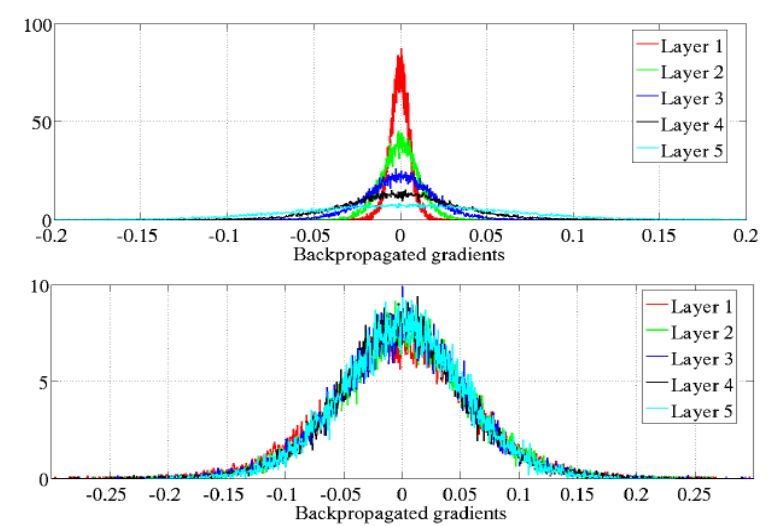


Figure 7: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.

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:

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

Initial Weight Normalization

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

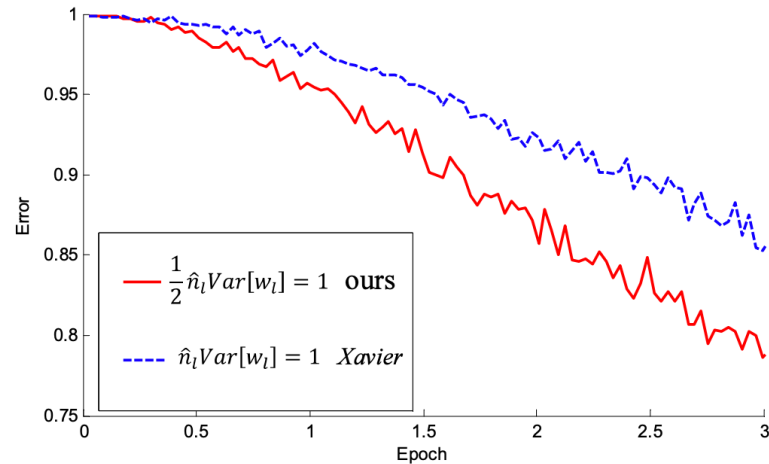


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and “Xavier” (blue)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

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

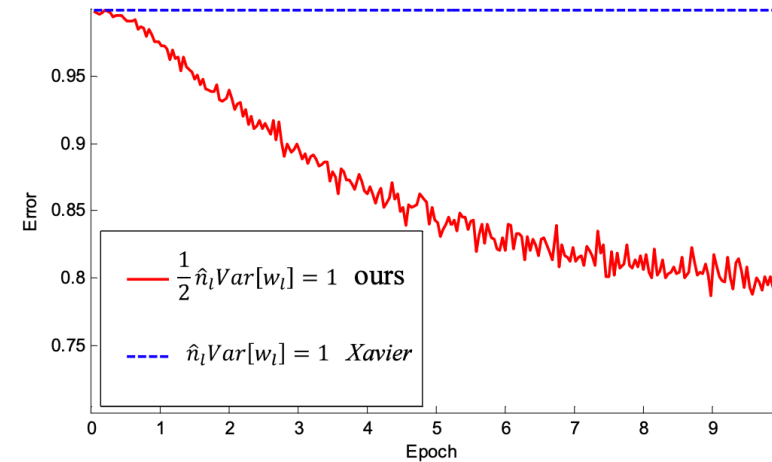


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But “Xavier” (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even 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: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β

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

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{ scale and shift}$$

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.

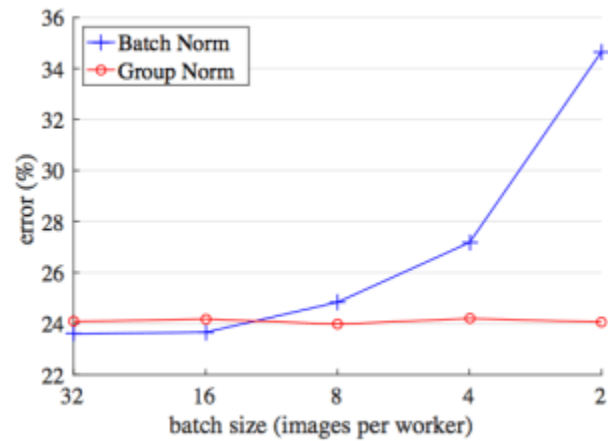
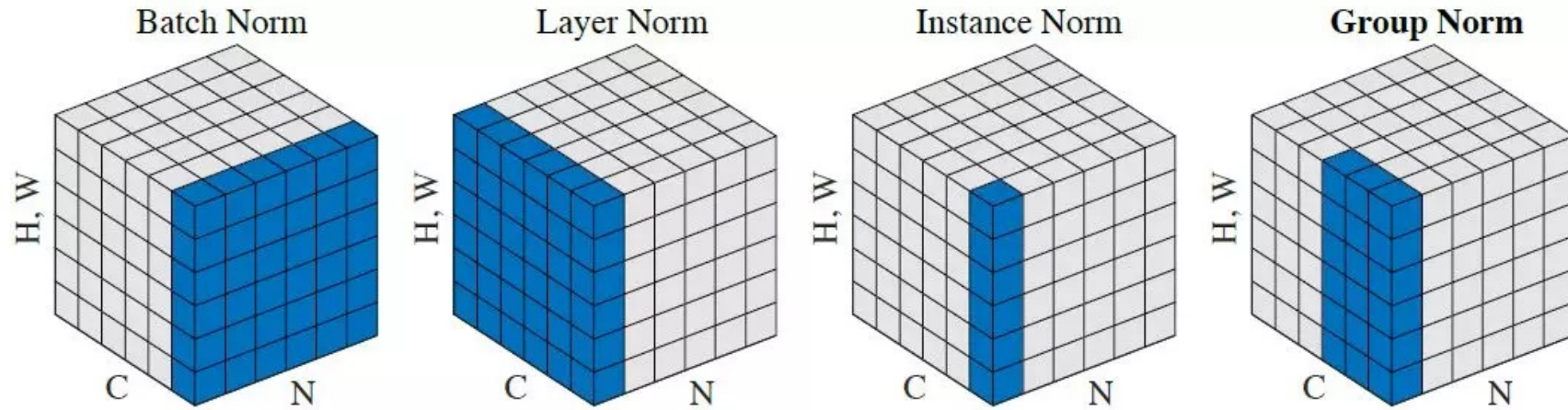
Alternative: Batch Normalization

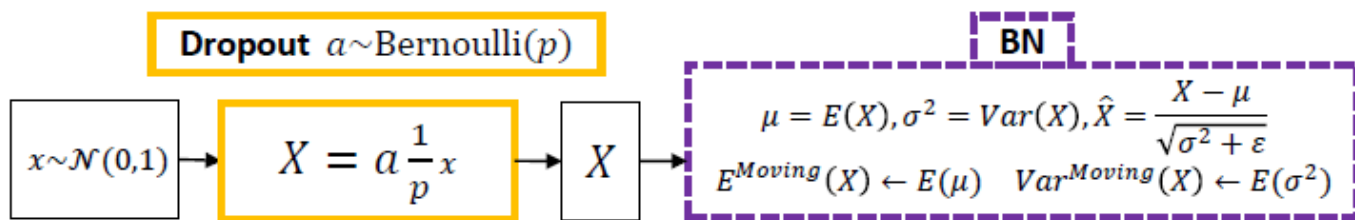
- Before or after non-linearity?
- Proposers
 - “If, however, **we could ensure that the distribution of nonlinearity inputs remains more stable** as the network trains, then the optimizer would be less likely to get stuck in the saturated regime, and the training would accelerate.”
 - “As each layer 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” (Hyvärinen & 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/>

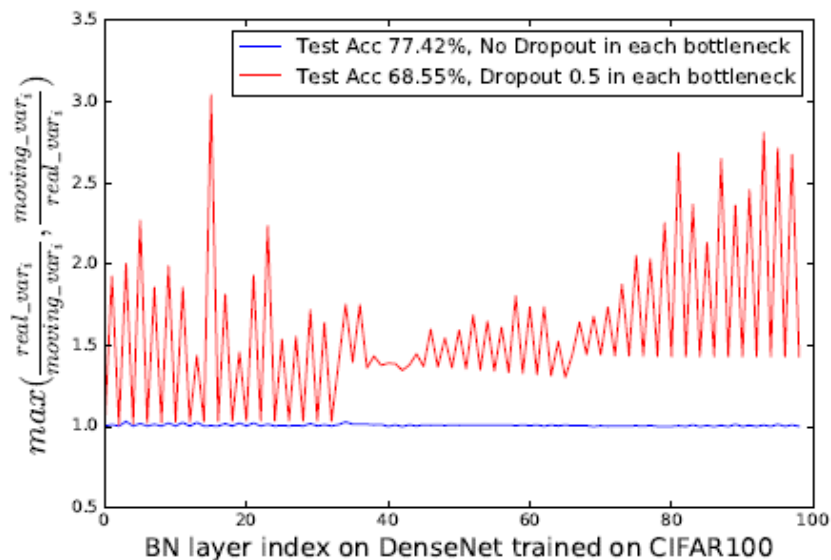
Alternative Normalizations





Train Mode $\text{Var}^{\text{Train}}(X) = \frac{1}{p} \rightarrow \text{Var}^{\text{Moving}}(X) = E\left(\frac{1}{p}\right)$

Test Mode $\text{Var}^{\text{Test}}(X) = 1 \not\rightarrow \text{Var}^{\text{Moving}}(X) = E\left(\frac{1}{p}\right)$



Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift

Xiang Li¹ Shuo Chen¹ Xiaolin Hu² Jian Yang¹

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?

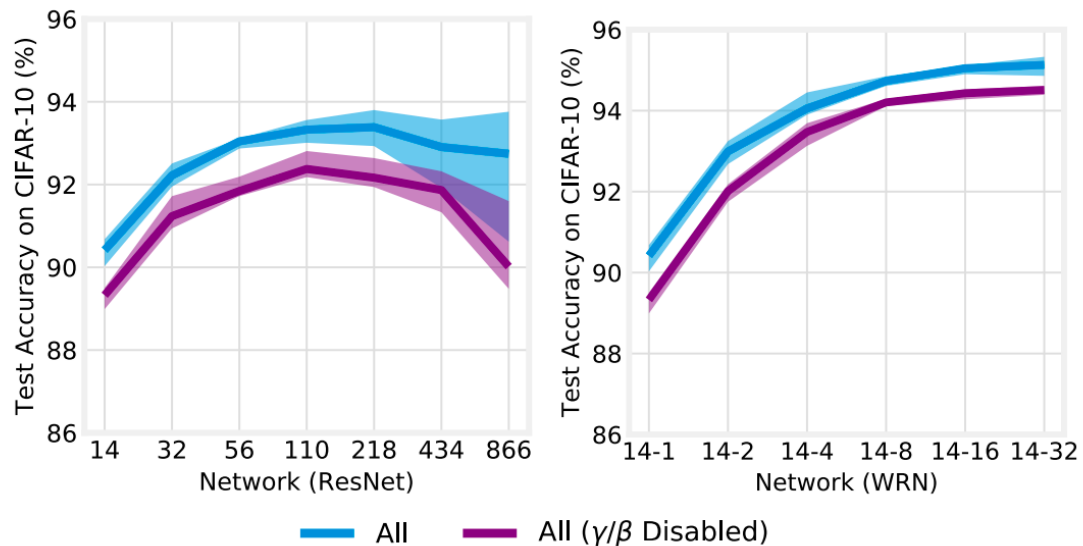


Figure 1. Test accuracy when training all parameters of the deep (left) and wide (right) ResNets in Table 1 with γ and β enabled (blue) and frozen at their initial values (purple). Except on the deepest ResNets, accuracy is about half a percent lower when γ and β are disabled.

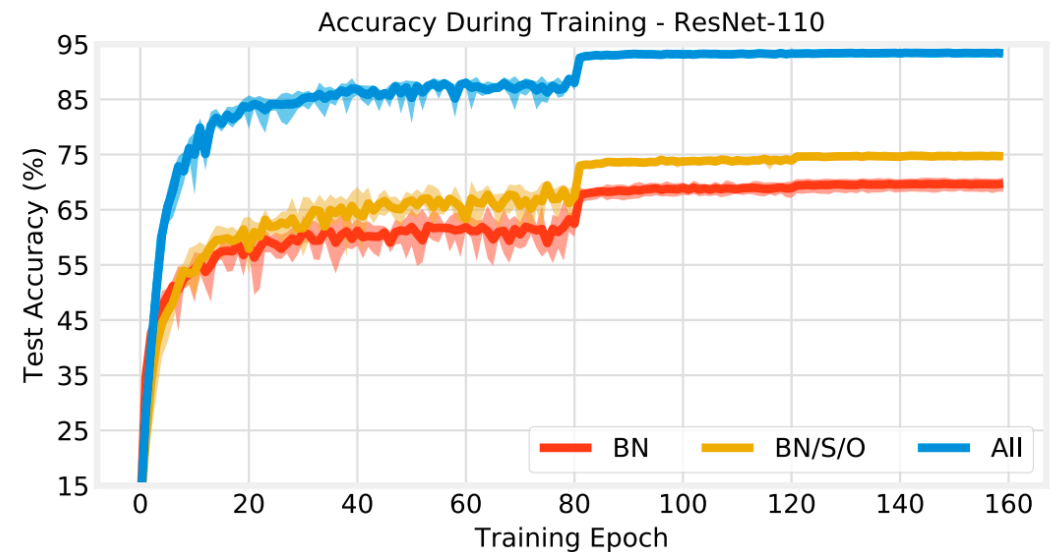


Figure 3. Test accuracy of ResNet-110 during training when training all parameters, just BatchNorm, and BatchNorm with shortcut and output parameters. Learning appears to occur at a similar rate in all experiments, although they reach different accuracies.

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

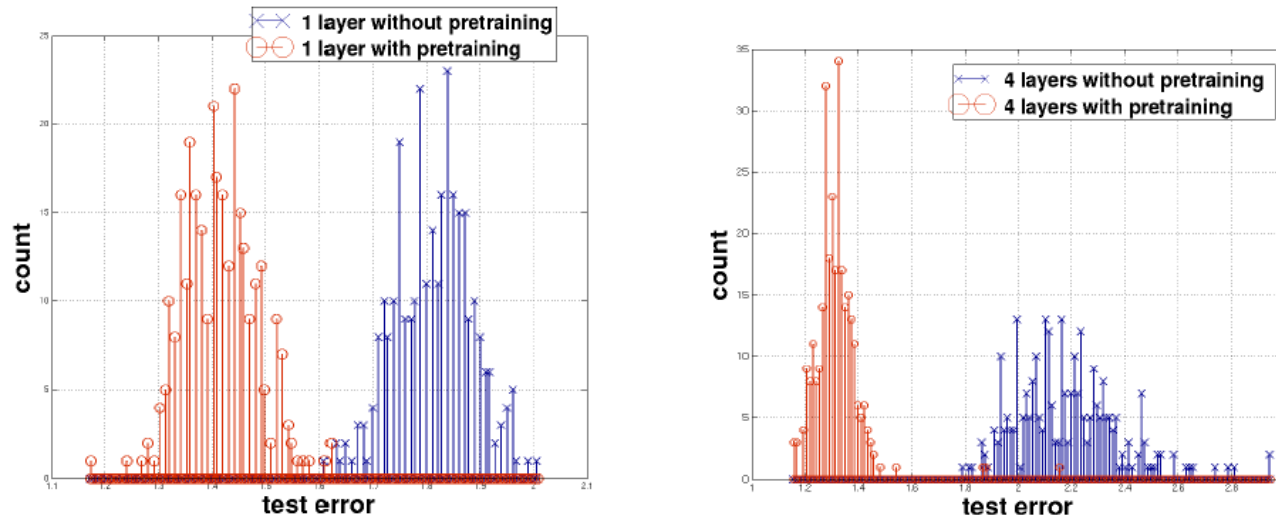


Figure 2: Histograms presenting the test errors obtained on MNIST using models trained with or without pre-training (400 different initializations each). **Left:** 1 hidden layer. **Right:** 4 hidden layers.

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*

Yoshua Bengio

Aaron Courville

Pierre-Antoine Manzagol

Pascal Vincent

Département d'informatique et de recherche opérationnelle

Université de Montréal

2920, chemin de la Tour

Montréal, Québec, H3T 1J8, Canada

Samy Bengio

Google Research

1600 Amphitheatre Parkway

Mountain View, CA, 94043, USA

DUMITRU.ERHAN@UMONTREAL.CA

YOSHUA.BENGIO@UMONTREAL.CA

AARON.COURVILLE@UMONTREAL.CA

PIERRE-ANTOINE.MANZAGOL@UMONTREAL.CA

PASCAL.VINCENT@UMONTREAL.CA

BENGIO@GOOGLE.COM

Unsupervised pretraining

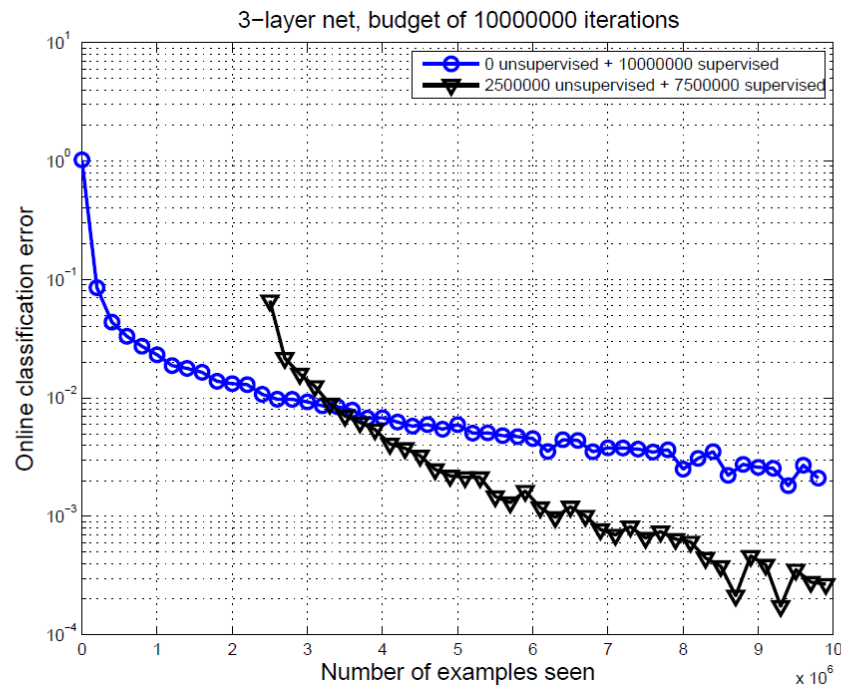


Figure 7: Deep architecture trained online with 10 million examples of digit images, either with pre-training (triangles) or without (circles). The classification error shown (vertical axis, log-scale) is computed online on the next 1000 examples, plotted against the number of examples seen from the 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

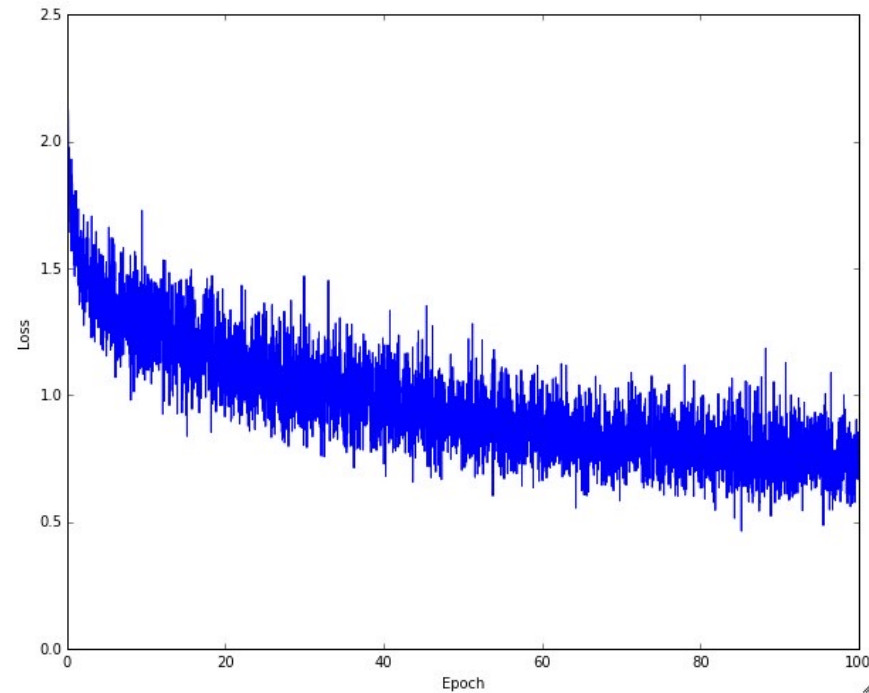
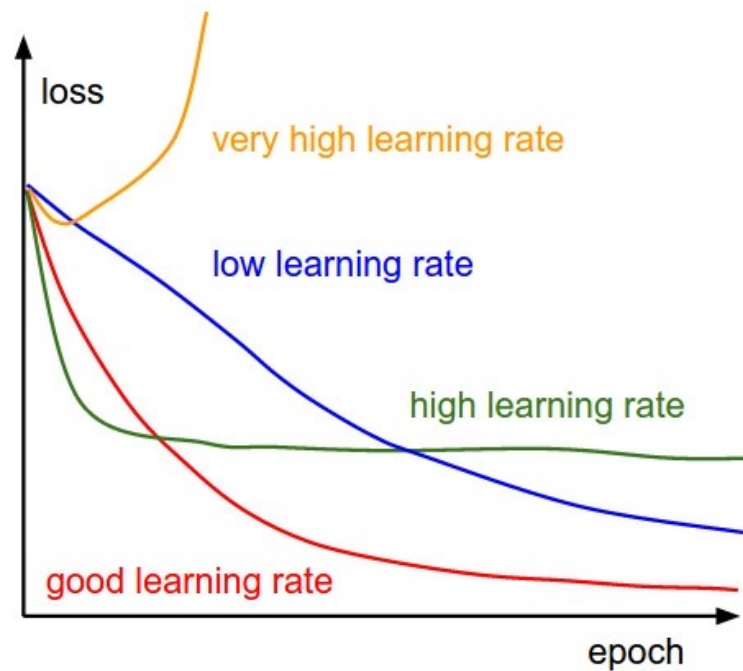
What if things are not working?

- 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?

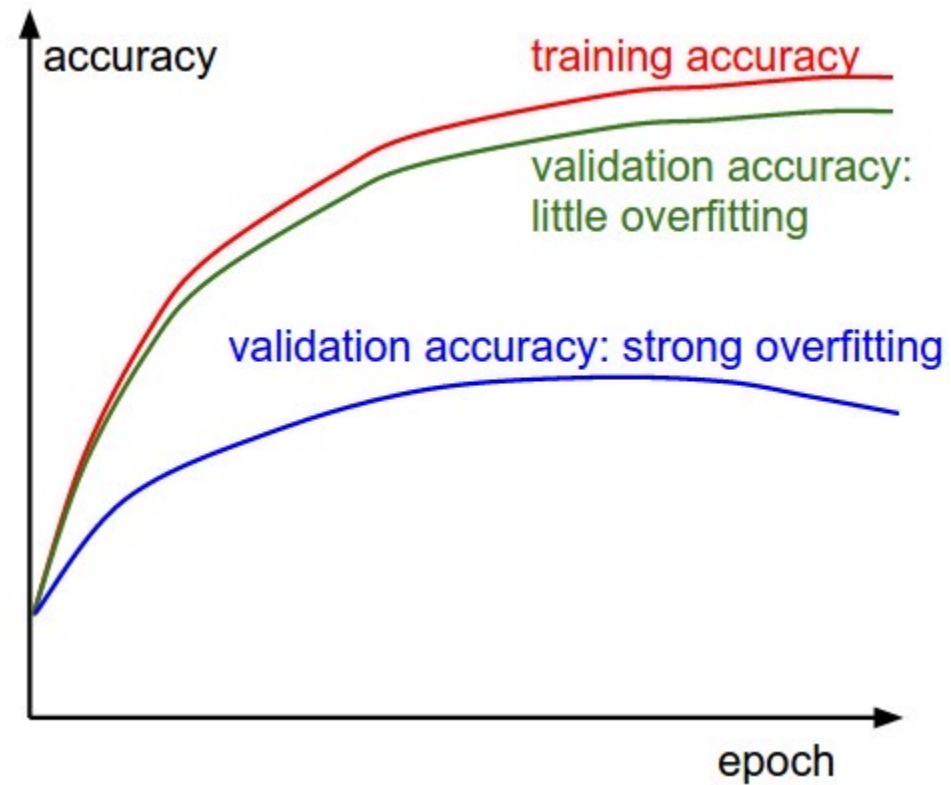
- Have a feeling of the initial loss value
 - For CIFAR-10 with 10 classes: because each class has probability of 0.1, initial loss is $-\ln(0.1)=2.302$
 - For hinge loss: since all margins are violated (since all scores are approximately zero), loss should be around 9 (+1 for each margin).
- Try to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

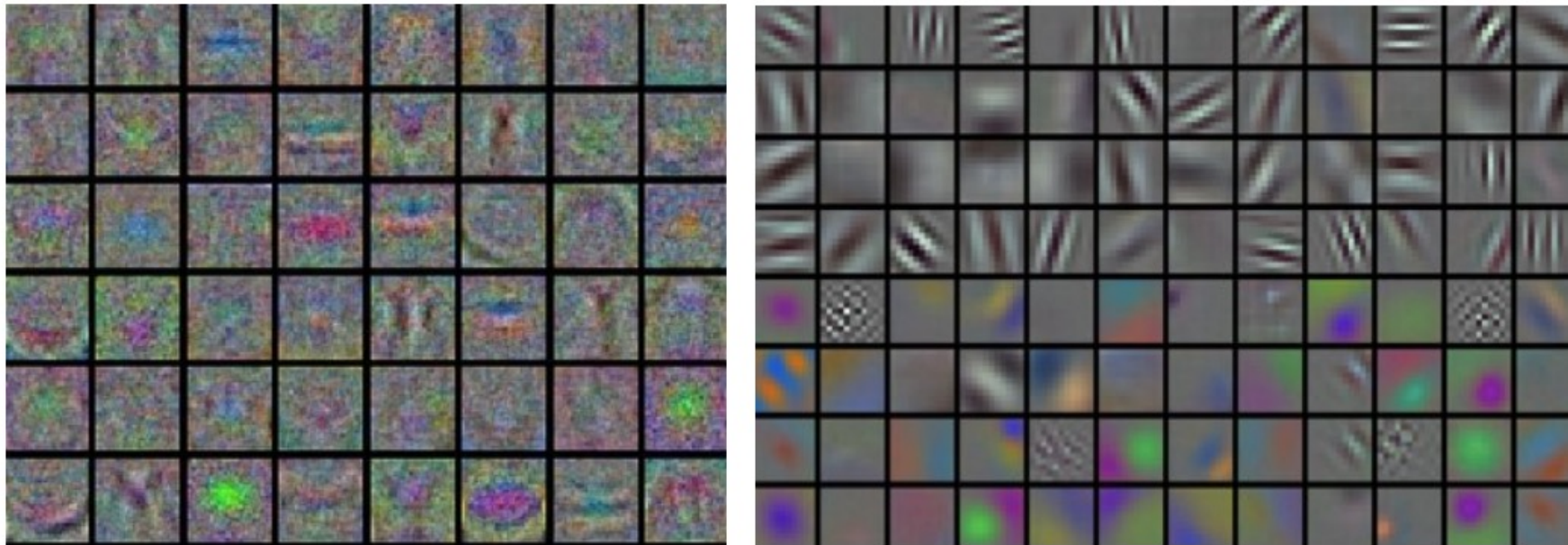


What if things are not working?

- 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)

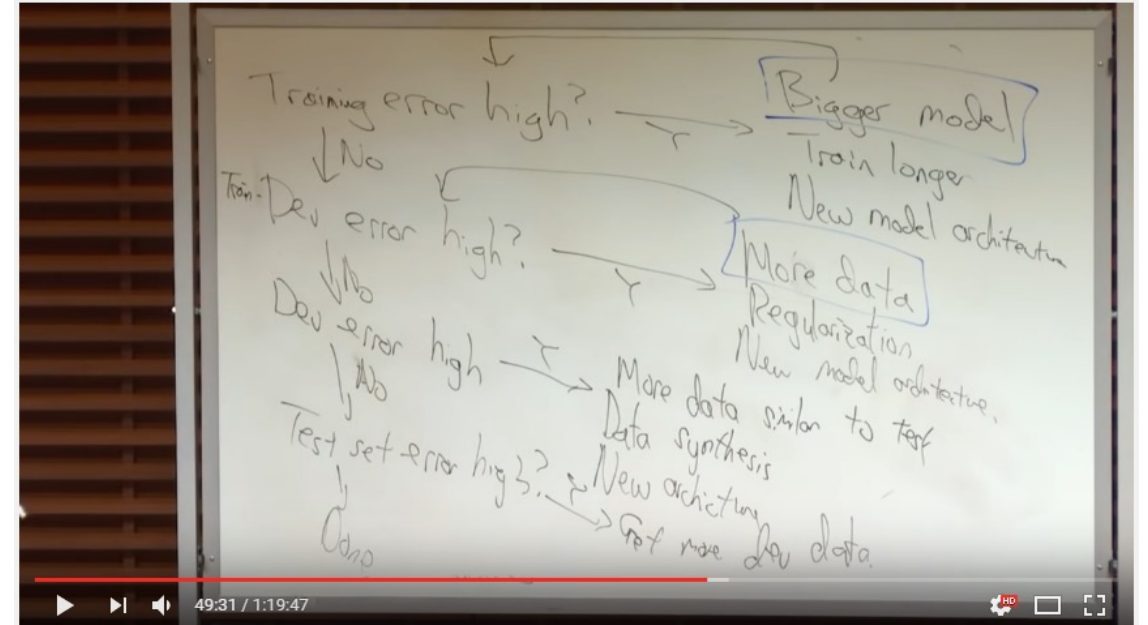
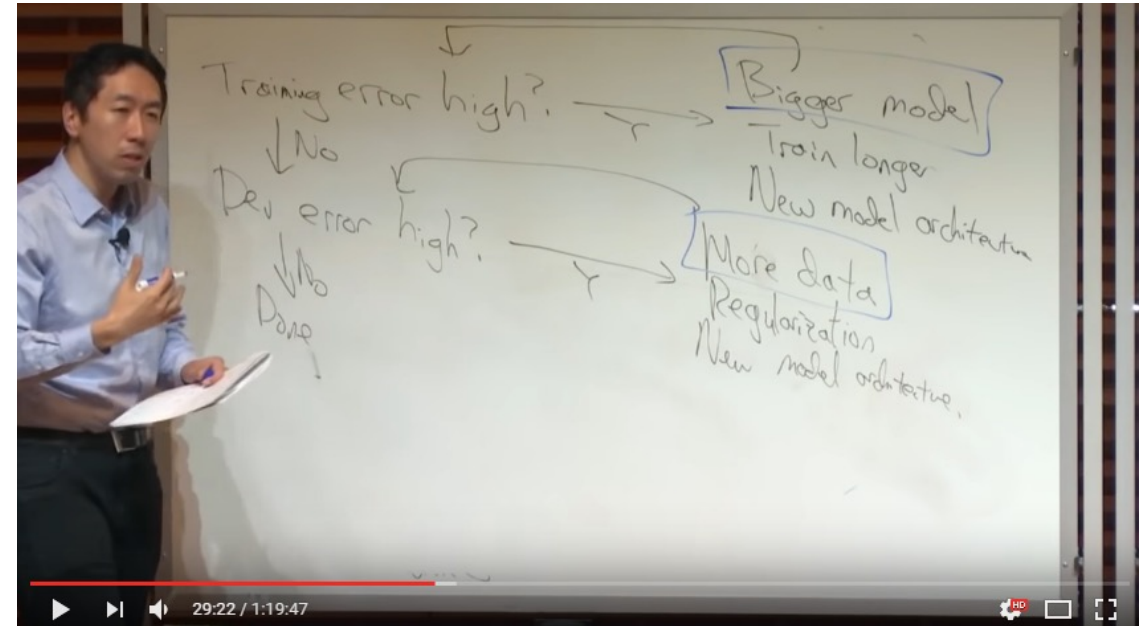


Examples of visualized weights for the first layer of a neural network. **Left:** Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty. **Right:** Nice, smooth, clean and diverse features are a good indication that the training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

Andrew Ng's suggestions

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

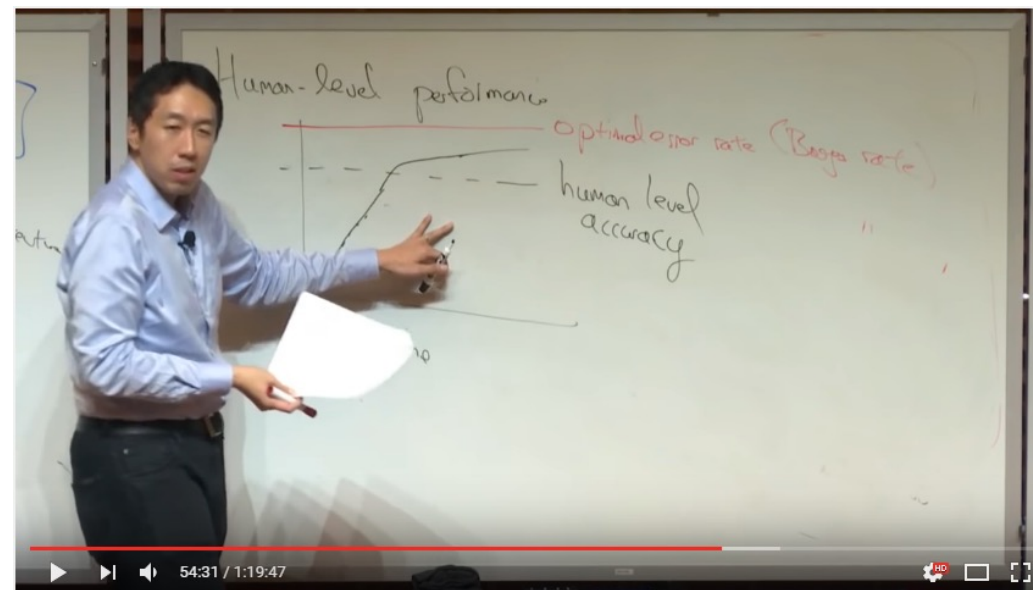
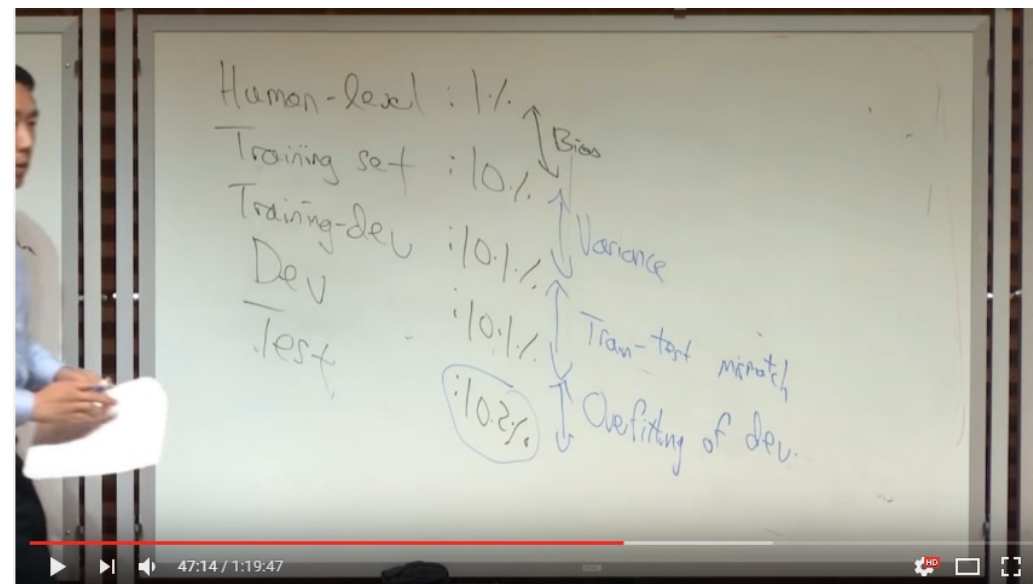
- “In DL, the coupling between bias & variance is weaker compared to other ML methods:
 - We can train a network to have high bias and variance.”
- “Dev (validation) and test sets should come from the same distribution. Dev&test sets are like problem specifications.
 - This requires especially attention 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

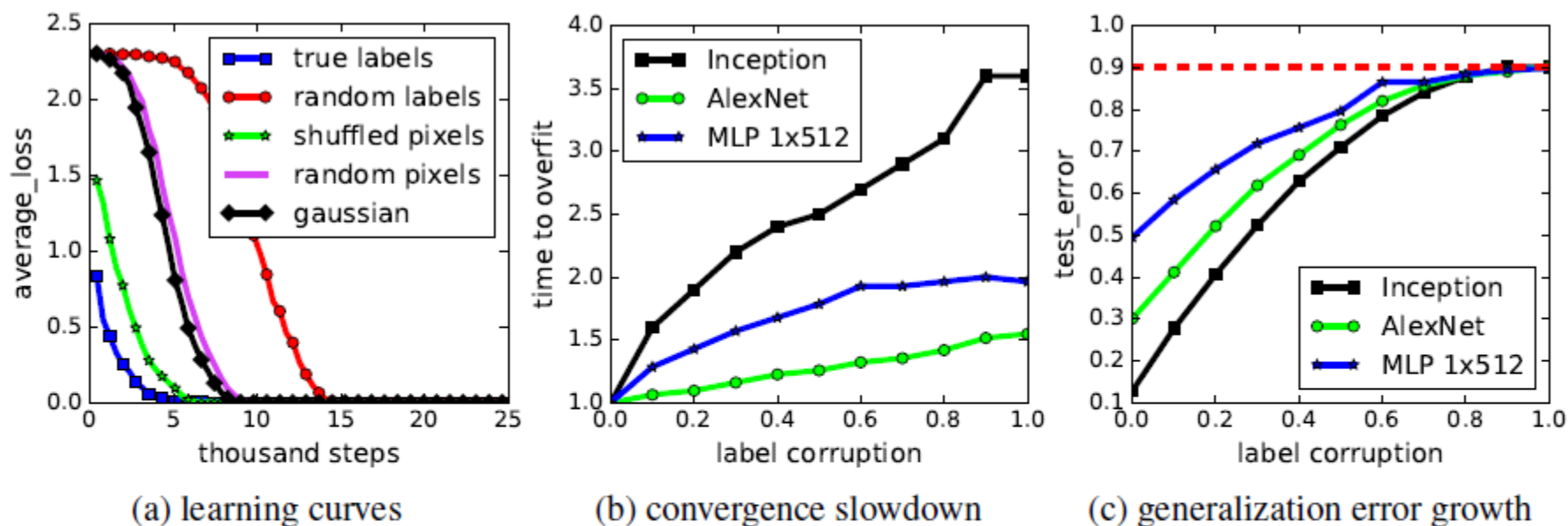


Figure 1: Fitting random labels and random pixels on CIFAR10. (a) shows the training loss of various experiment settings decaying with the training steps. (b) shows the relative convergence time with different label corruption ratio. (c) shows the test error (also the generalization error since training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