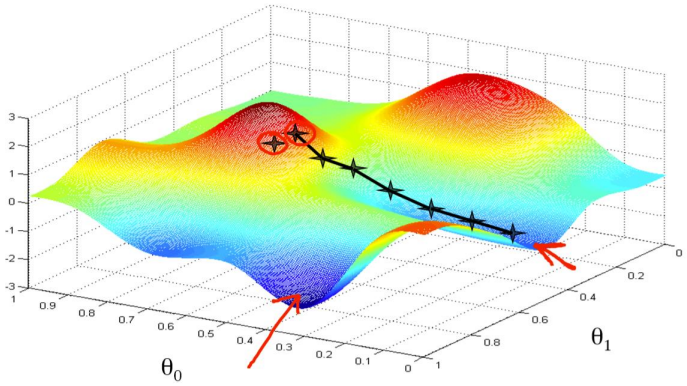
Deep Learning (COSE474)
2024 Fall

8. Training Neural Networks I

Hyunwoo J. Kim
(hyunwoojkim@korea.ac.kr)



Agenda

- Questions?
- Training NNs
  - Optimization
  - Stochastic Gradient Descent (SGD)
  - Mini-Batch SGD
- (1) Data preprocessing
- (2) Data augmentation
- (3) Architecture
  - Activation functions
- (4) Initialization
- (5) Code Sanity Check with a small dataset
- (6) Hyperparameter tuning: Find a learning rate, Regularization

Overview

1. One time setup

- activation functions, preprocessing, weight initialization, regularization,

2. Training dynamics

- babysitting the learning process, parameter updates

3. Evaluation

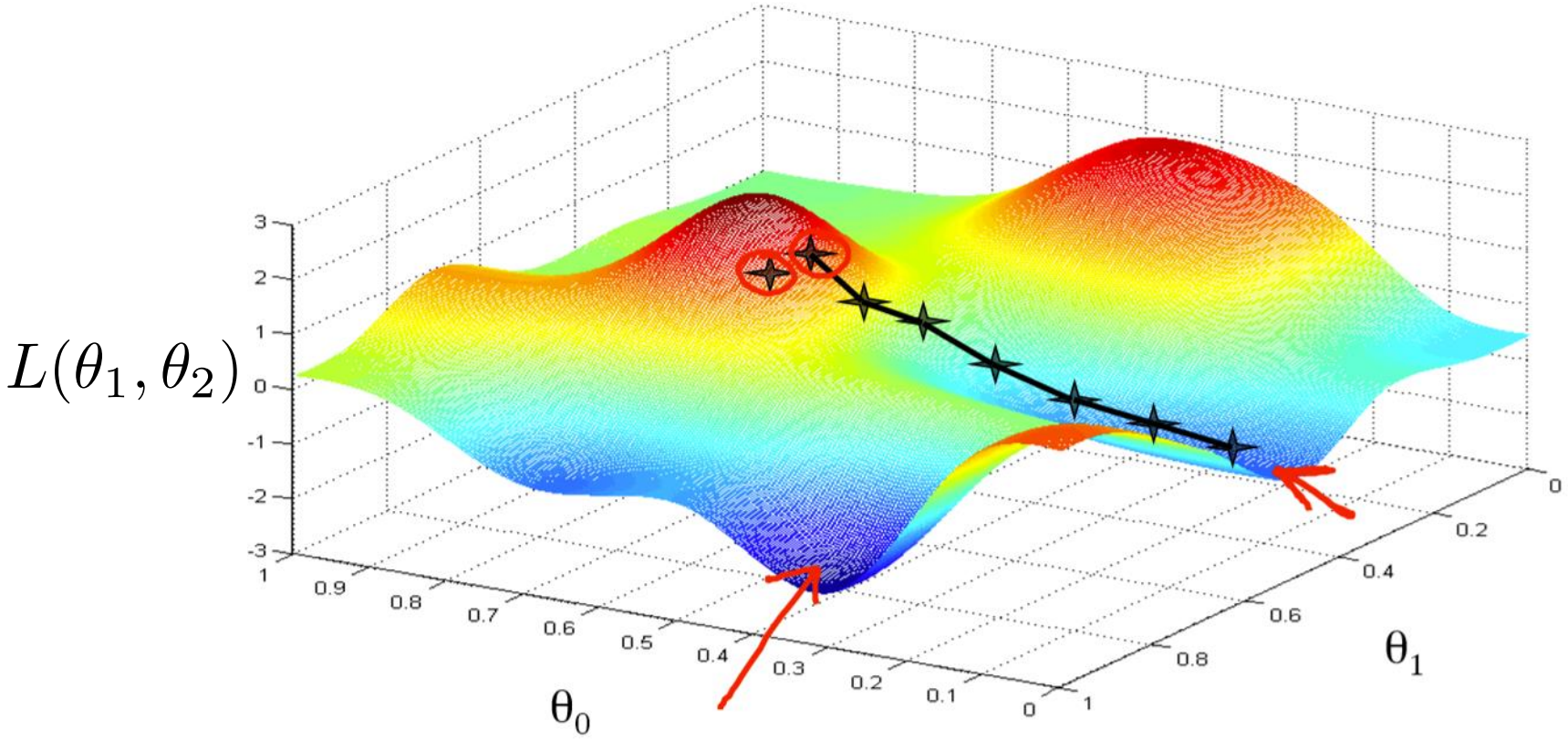
Training CNNs

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate, and regularization strength
- Minimize the loss and monitor progress
- Play with knobs

Readings

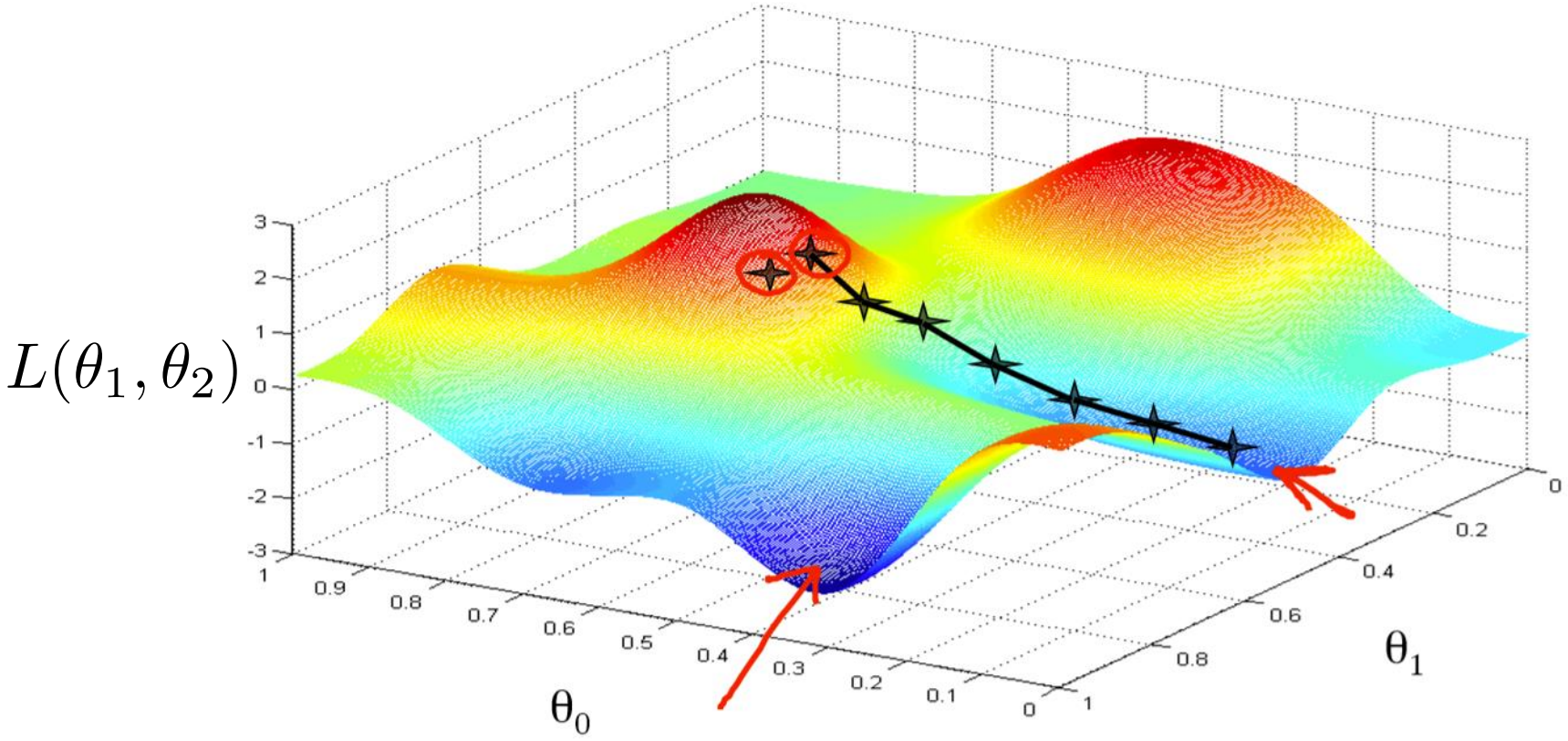
- PyTorch API Short Introduction from cs231n
  - http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture06.pdf
- Stochastic Gradient Descent & Backpropagation
  - http://cs231n.github.io/optimization-1/
  - http://cs231n.github.io/optimization-2/
- Best practices for training CNNs
  - http://cs231n.github.io/neural-networks-2/
  - http://cs231n.github.io/neural-networks-3/

Learning weights through optimization



$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta)$$

Learning weights through optimization



$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L(\theta)$$

Stochastic Gradient?

$$\mathbb{E}[\nabla \tilde{f}(x)] = \nabla f(x)$$

Stochastic Gradient?

$$\mathbb{E}[\nabla \tilde{f}(x)] = \nabla f(x)$$

$$f = \frac{1}{N} \sum_{i=1}^{N} f_i$$

$$\tilde{f} = f_i \text{ with } i \text{ from uniform distr. } \{1, \ldots, N\}$$

Mini-batch Stochastic Gradient

$$\mathbb{E}[\nabla \tilde{f}(x)] = \nabla f(x)$$

$$f = \frac{1}{N} \sum_{i=1}^{N} f_i$$

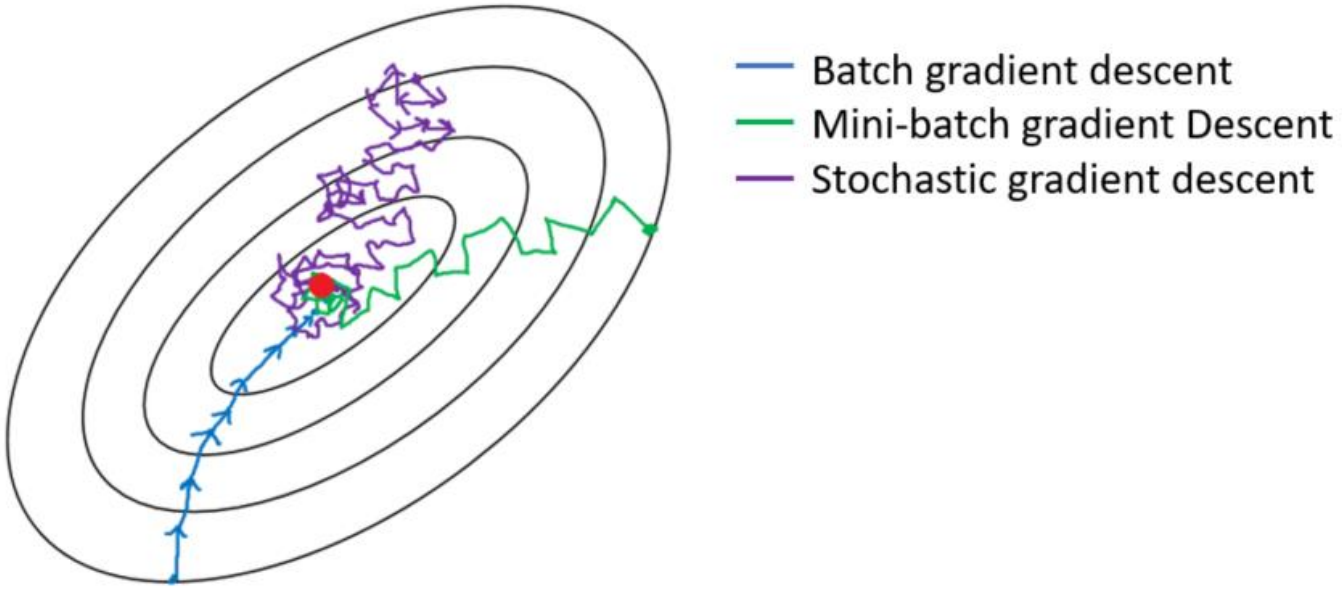
$$f = \frac{1}{|K|} \sum_{i \in K} f_i, \text{ where } K \text{ is a set of random } k \text{ indices } \{1, \ldots, N\}$$

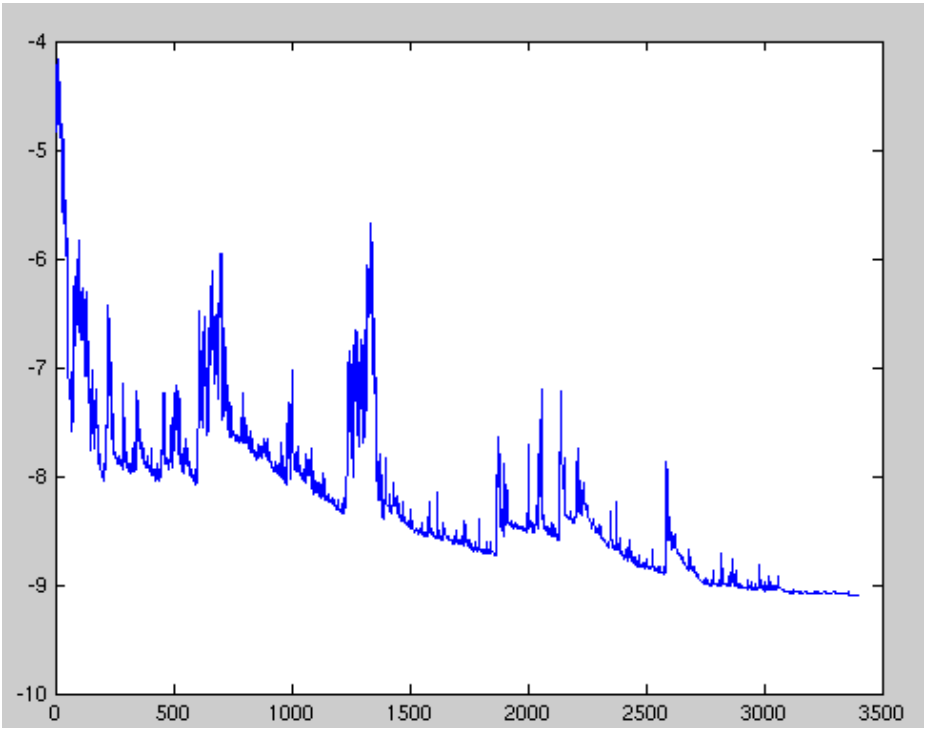
Mini-batch SGD

Loop:

1. Sample a batch of data (batch size?)
2. Forward prop it through the graph (network), get loss
3. Backprop to calculate the gradients
4. Update the parameters using the gradient

GD, SGD, mini-batch SGD



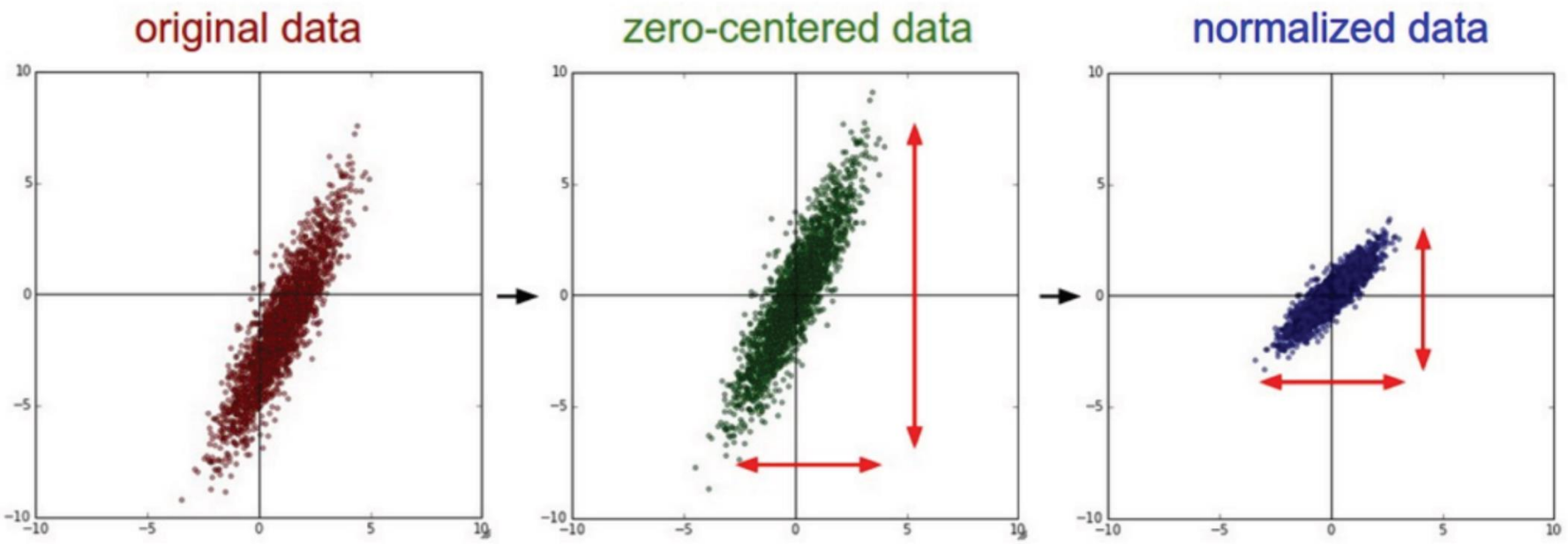


Img src: https://medium.com/analytics-vidhya/gradient-descent-vs-stochastic-gd-vs-mini-batch-sgd-fbd3a2cb4ba4

SGD learning curve

(1) Data preprocessing

Preprocess the data so that learning is better conditioned:



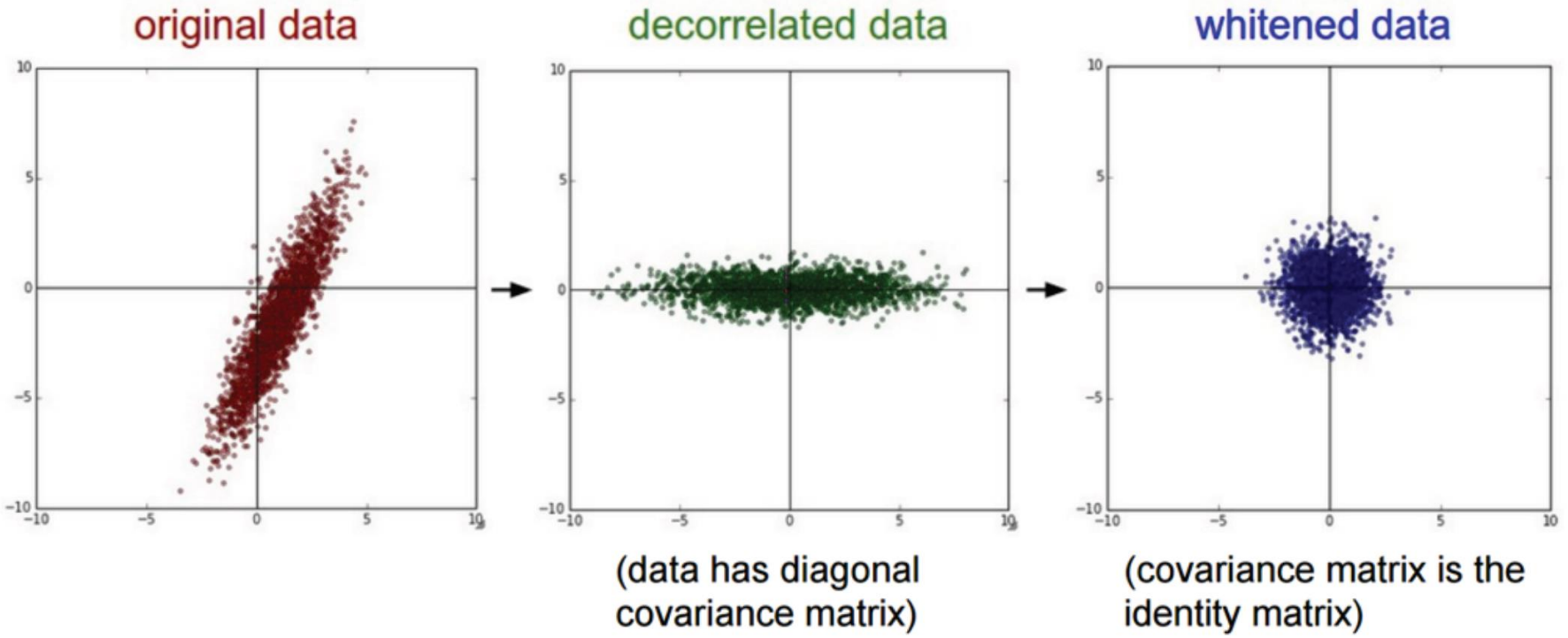

```
X -= np.mean(axis=0, keepdims=True)
```

```
X /= np.std(axis=0, keepdims=True)
```

Figure: Andrej Karpathy

(1) Data preprocessing

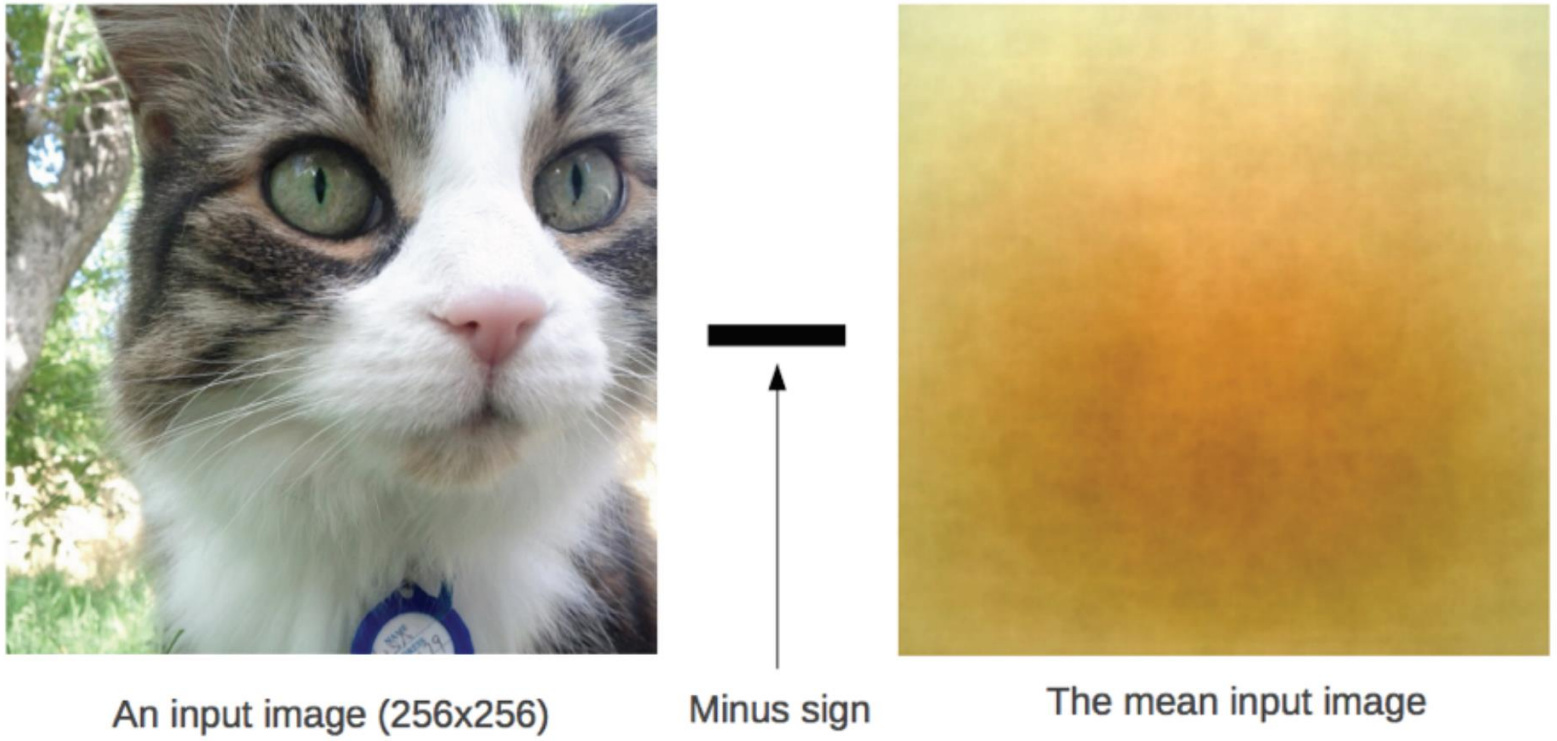
In practice, you may also see **PCA** and **Whitening** of the data:



Slide: Andrej Karpathy

(1) Data preprocessing

For ConvNets, typically only the mean is subtracted.



A per-channel mean also works (one value per R,G,B).

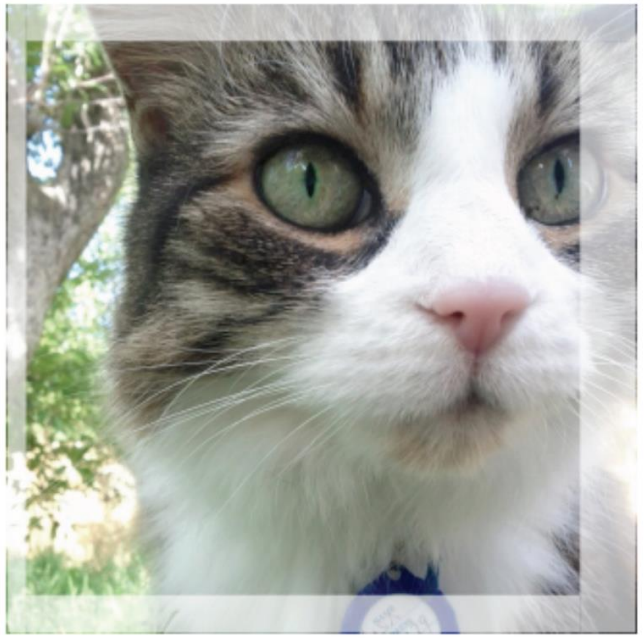
Figure: Alex Krizhevsky

TLDR: In practice for Images: center only

- Subtract the mean image (e.g. AlexNet)
  - (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  - (mean along each channel = 3 numbers)
- Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
  - (mean along each channel = 3 numbers)

(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (e.g. [Krizhevsky 2012]) overfit the data.



E.g. 224x224 patches extracted from 256x256 images

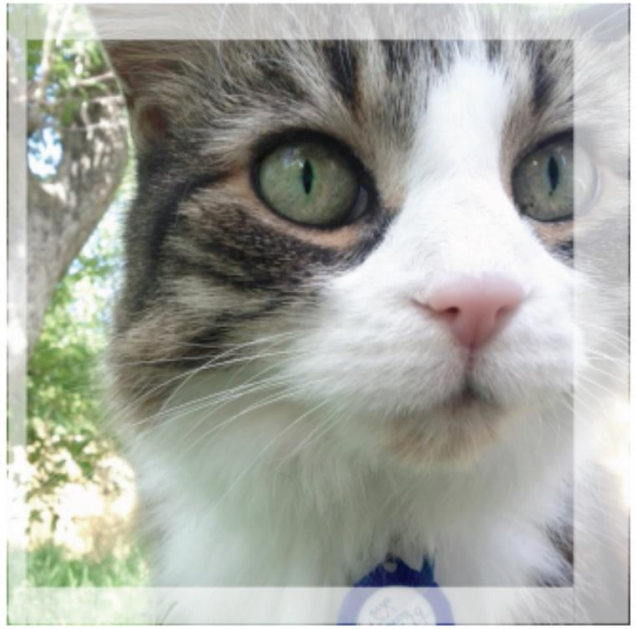
Randomly reflect horizontally

Perform the augmentation live during training

Figure: Alex Krizhevsky

(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (e.g. [Krizhevsky 2012]) overfit the data.



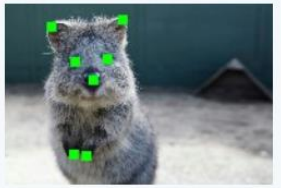
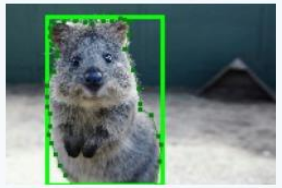
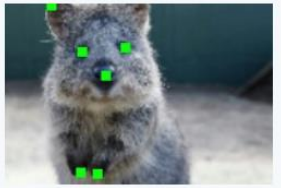
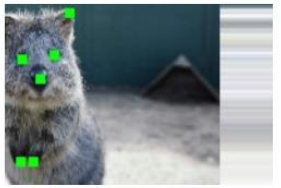
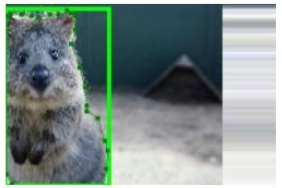
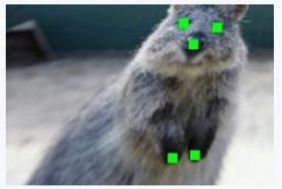
E.g. 224x224 patches extracted from 256x256 images

Randomly reflect horizontally

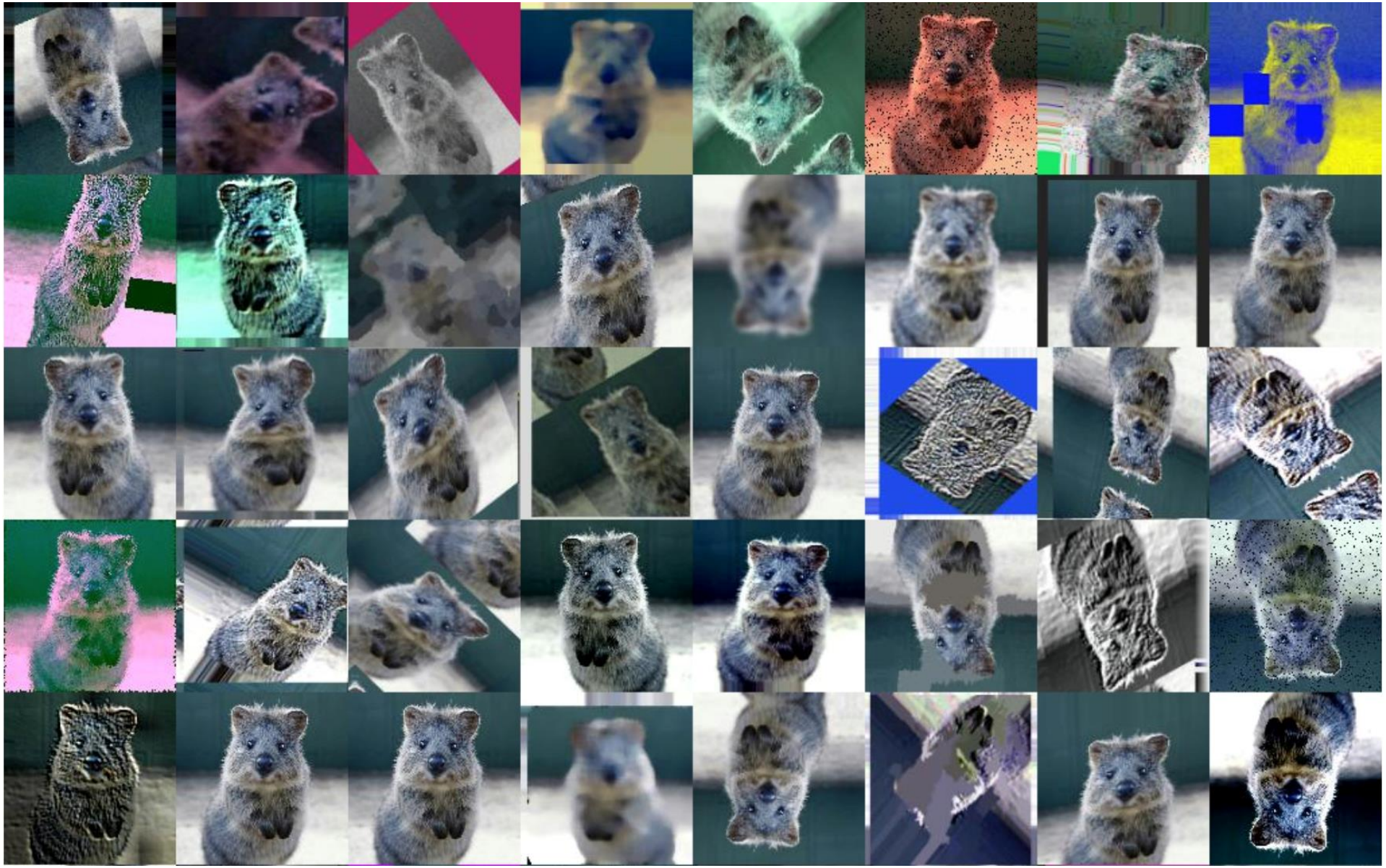
Perform the augmentation live during training

Figure: Alex Krizhevsky

Data Augmentation

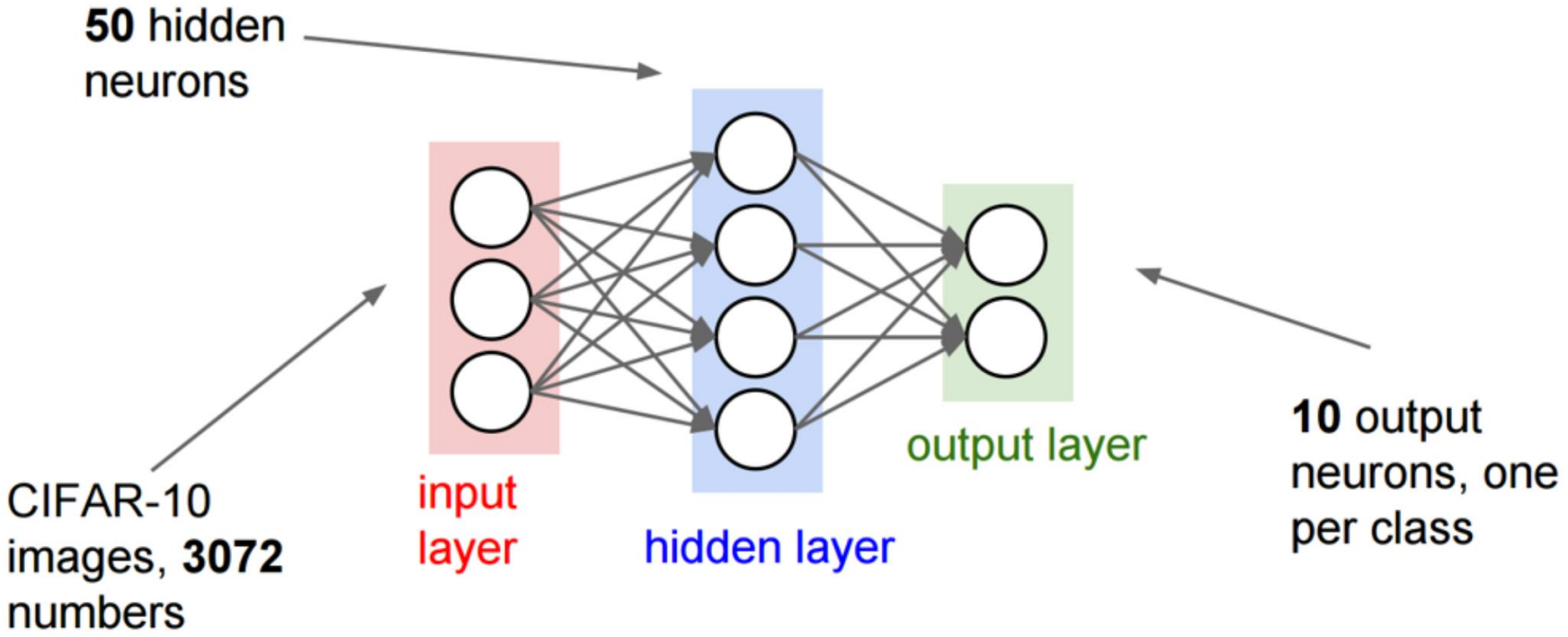
	Image	Heatmaps	Seg. Maps	Keypoints	Bounding Boxes, Polygons
Original Input					
Gauss. Noise + Contrast + Sharpen					
Affine					
Crop + Pad					
Fliplr + Perspective					

Learn functions work well on all the augmented data rather than finding Invariant features.



(2) Choose your architecture

Toy example: one hidden layer of size 50



Slide: Andrej Karpathy

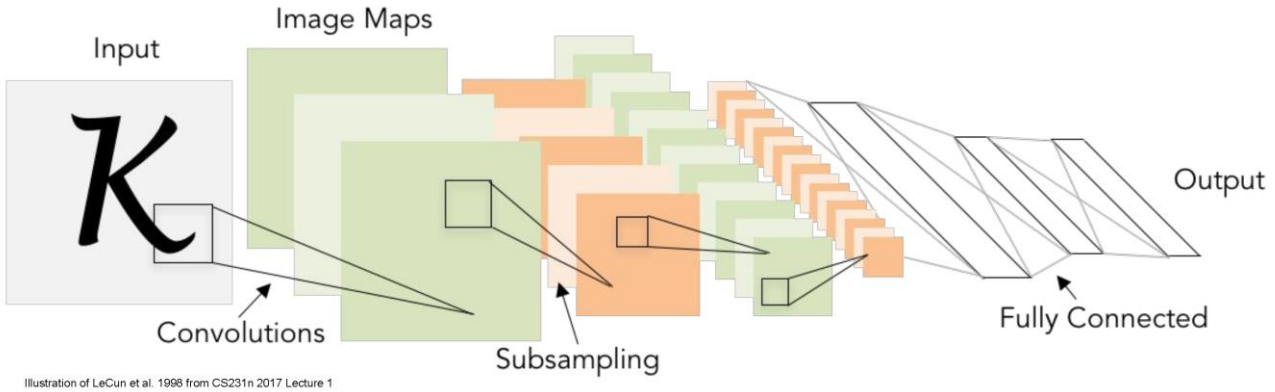
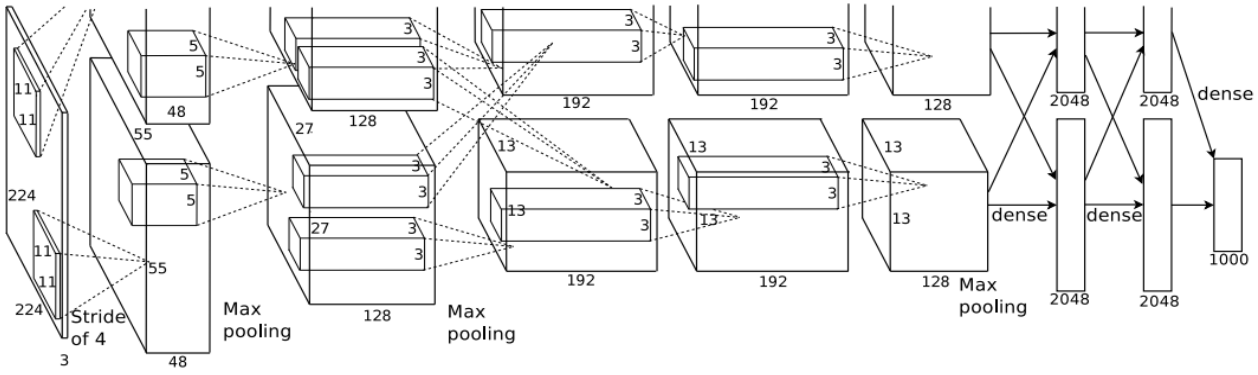
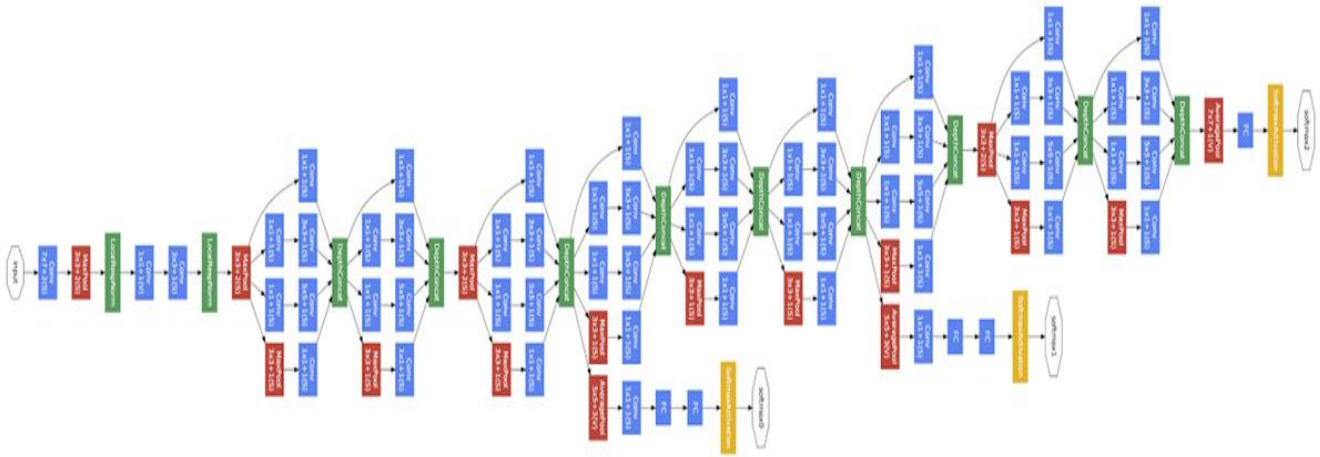
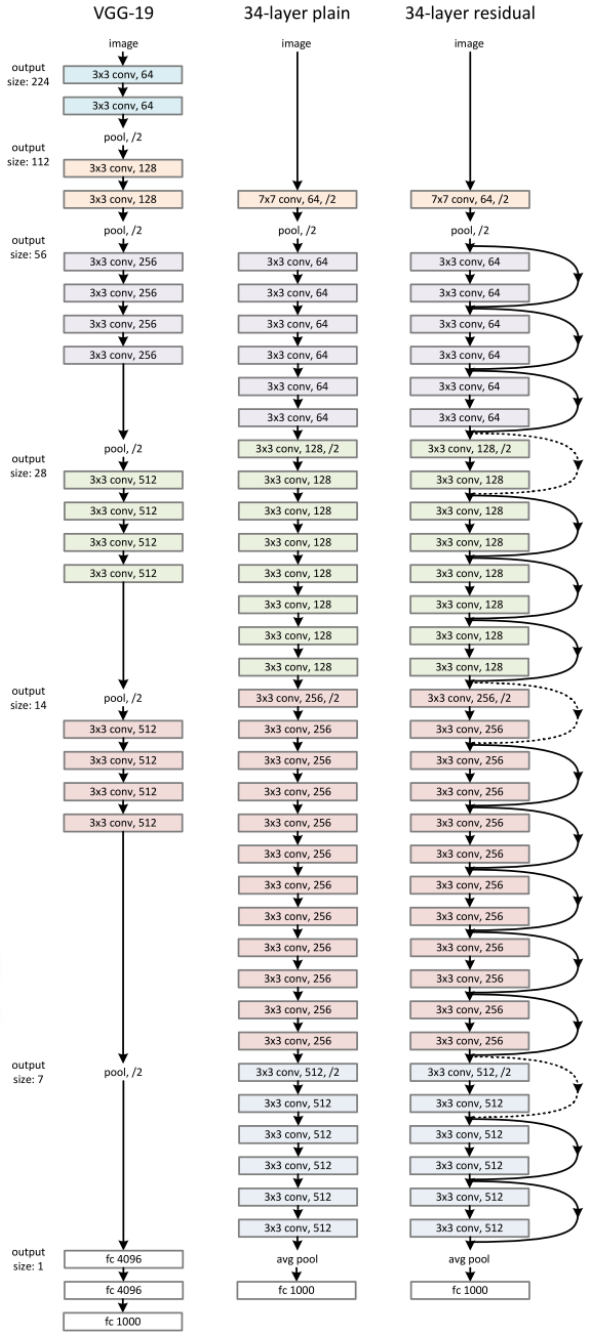


Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1



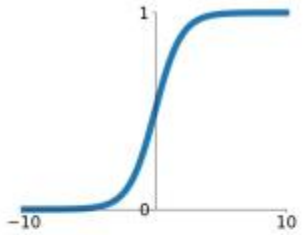




Activation Functions

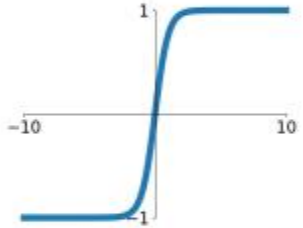
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



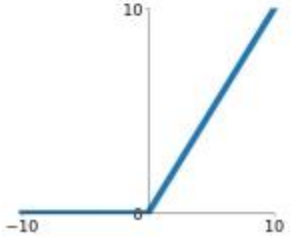
tanh

$$\tanh(x)$$



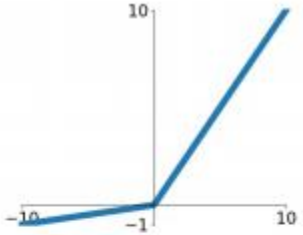
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

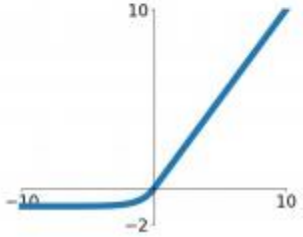


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

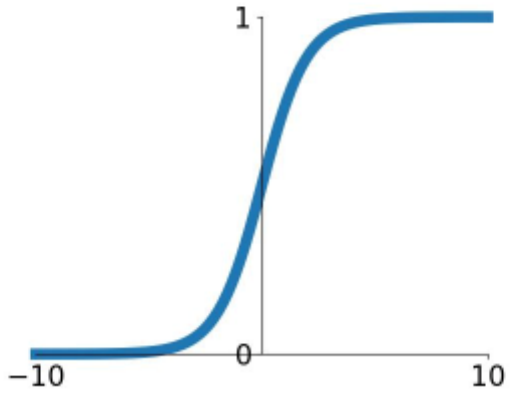
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions

Activation Functions



Sigmoid

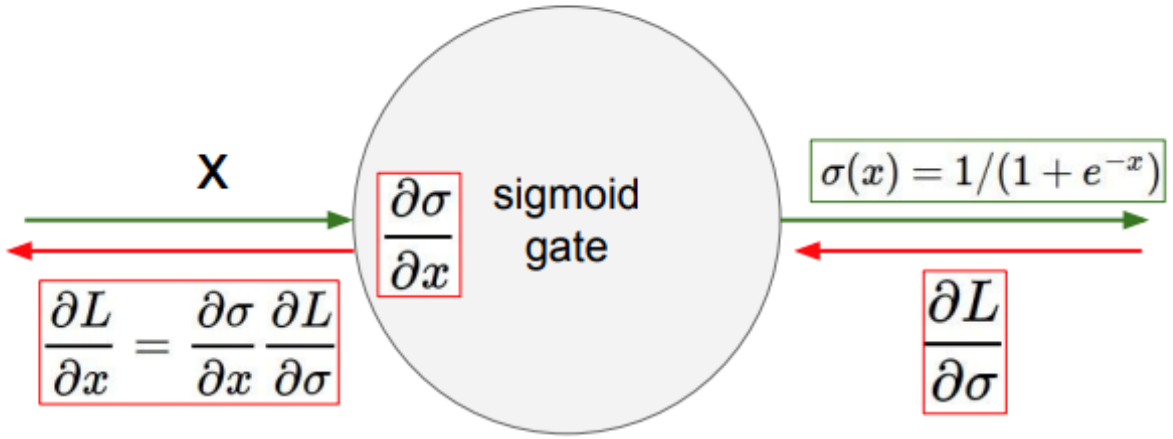
$$\sigma(x) = 1/(1 + e^{-x})$$

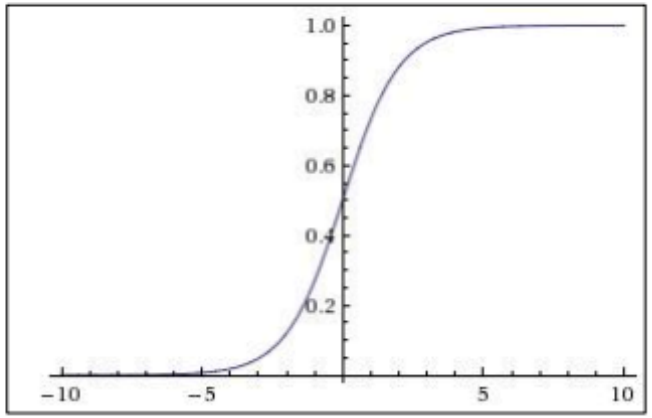
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients

Activation Functions

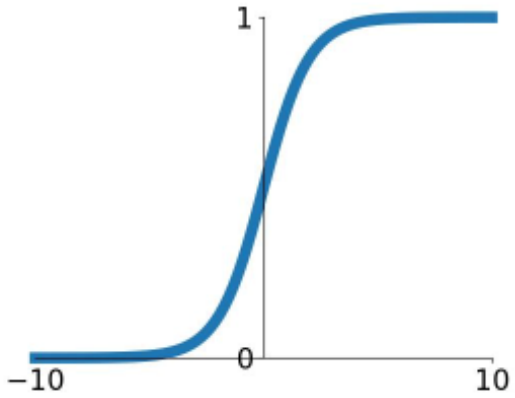




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Sigmoid

Activation Functions



Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

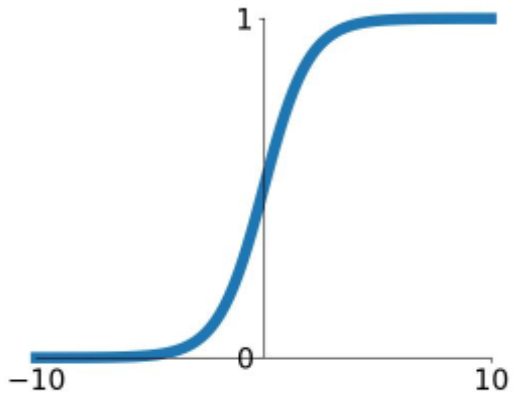
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

Sigmoid

Activation Functions



Sigmoid

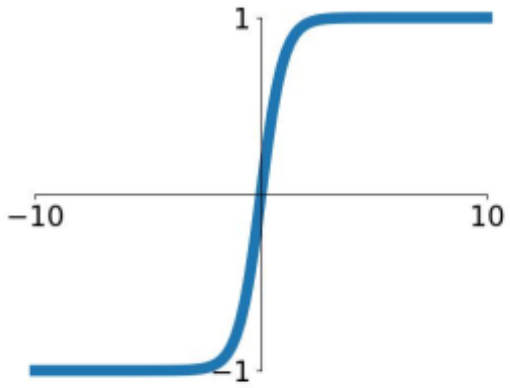
$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

tanh

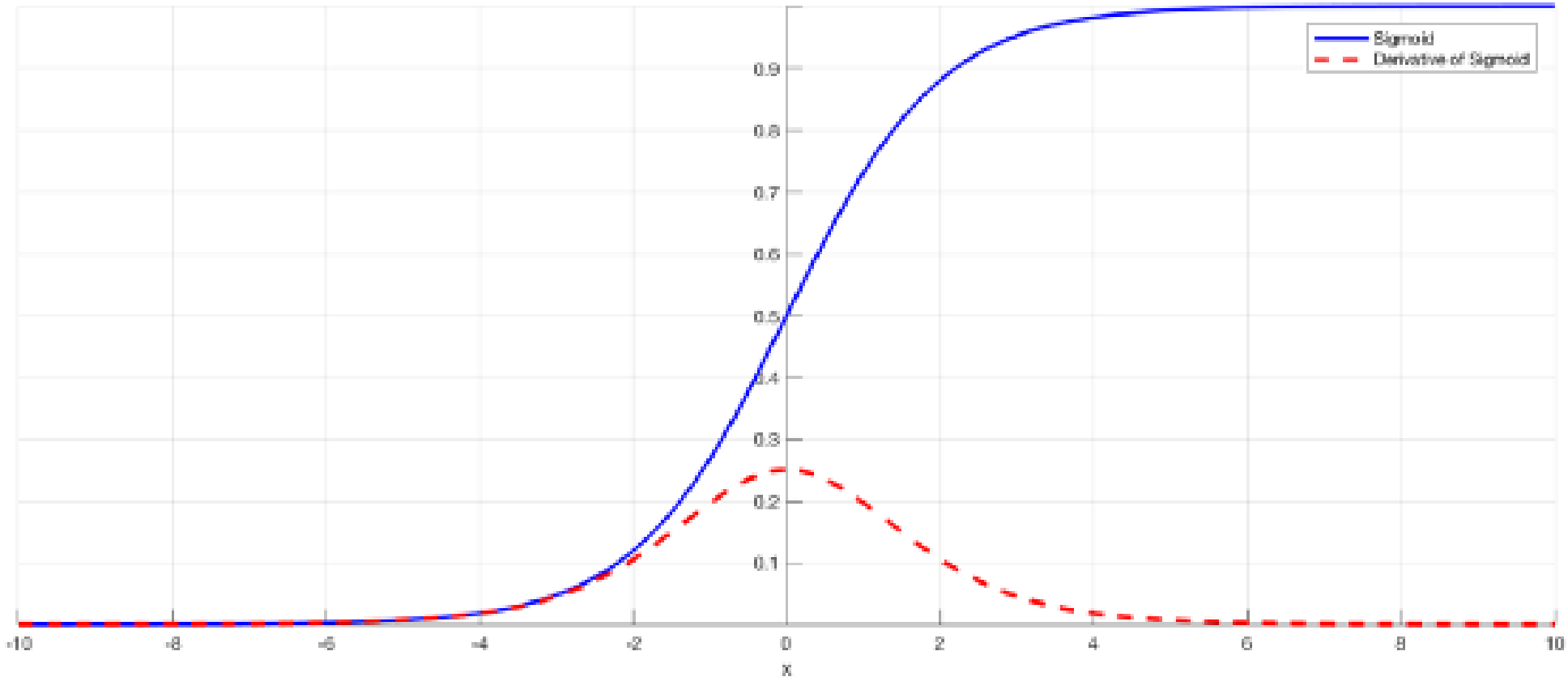


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

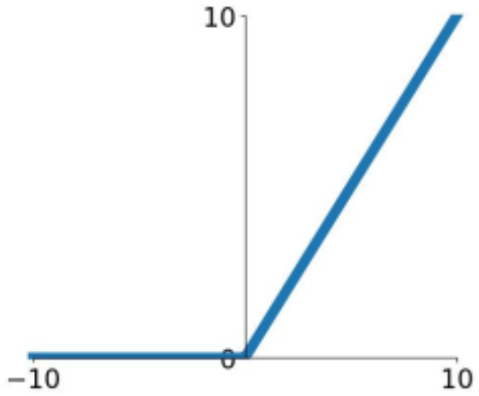
Sigmoid and its derivative



https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

Rectified Linear Unit (ReLU)

Activation Functions



ReLU
(Rectified Linear Unit)

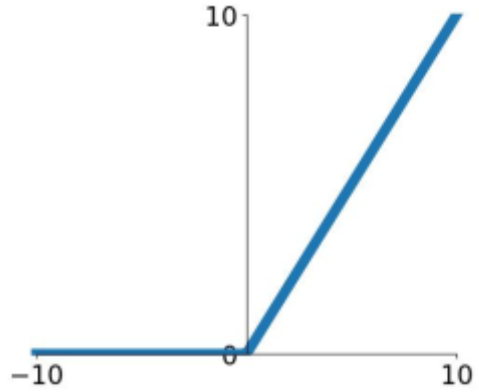
- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

ReLU (Rectified Linear Unit)

Activation Functions



ReLU
(Rectified Linear Unit)

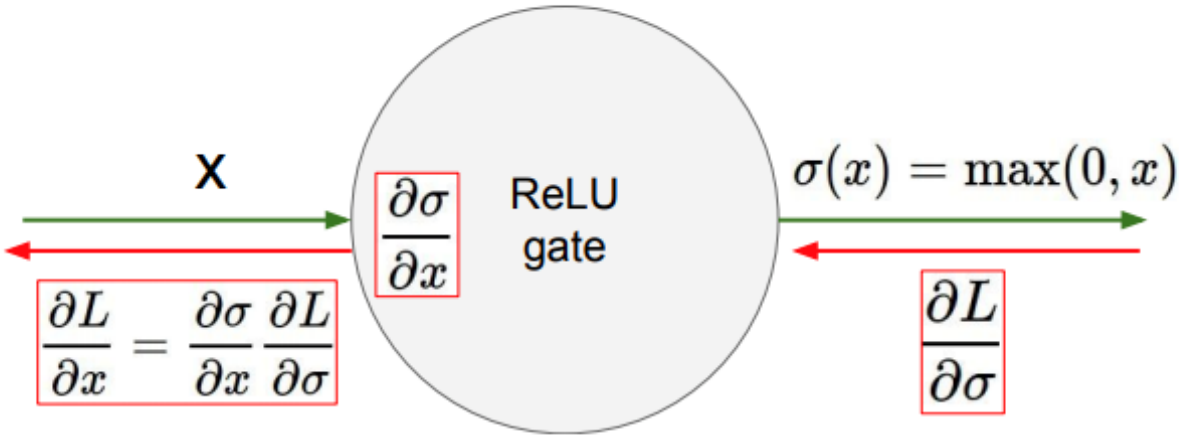
- Computes **f(x) = max(0,x)**

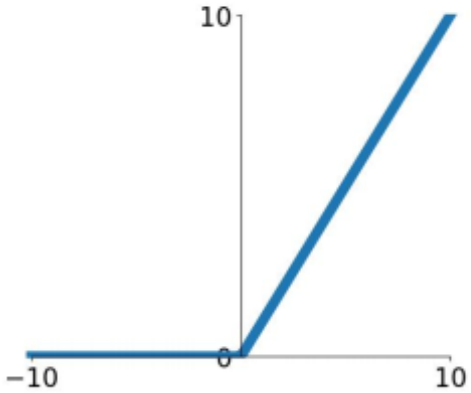
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

ReLU



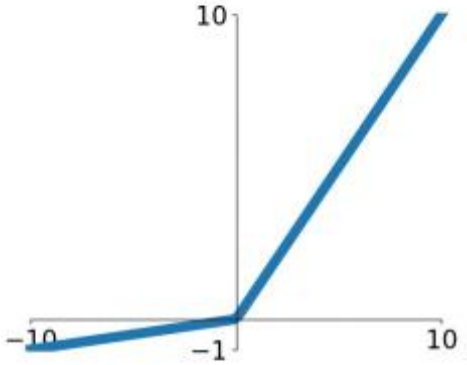


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Leaky ReLU

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid for very deep neural networks

Initialization

Set the weights to small random numbers:

```
W = np.random.randn(D, H) * 0.001
```

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

```
b = np.zeros(H)
```

Slide: Andrej Karpathy

Research on Initialization

- **Understanding the difficulty of training deep feedforward neural networks** by Glorot and Bengio, 2010
- **Exact solutions to the nonlinear dynamics of learning in deep linear neural networks** by Saxe et al, 2013
- **Random walk initialization for training very deep feedforward networks** by Sussillo and Abbott, 2014
- **Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification** by He et al., 2015
- **Data-dependent Initializations of Convolutional Neural Networks** by Krähenbühl et al., 2015
- **All you need is a good init**, Mishkin and Matas, 2015
- **Fixup Initialization: Residual Learning Without Normalization**, Zhang et al, 2019
- **The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks,** Frankle and Carbin, 2019

Sanity Check with a small dataset (overfitting)

- Check whether loss function values are correct
  - Turn on/off regularization, check predictions, loss values for each sample
- Check whether the model can achieve perfect accuracy
  - If a model is sufficiently large, it MUST be possible to fit a model to a small dataset perfectly.
- Check your code
  - For instance, whether it saves checkpoints since an epoch takes much shorter.

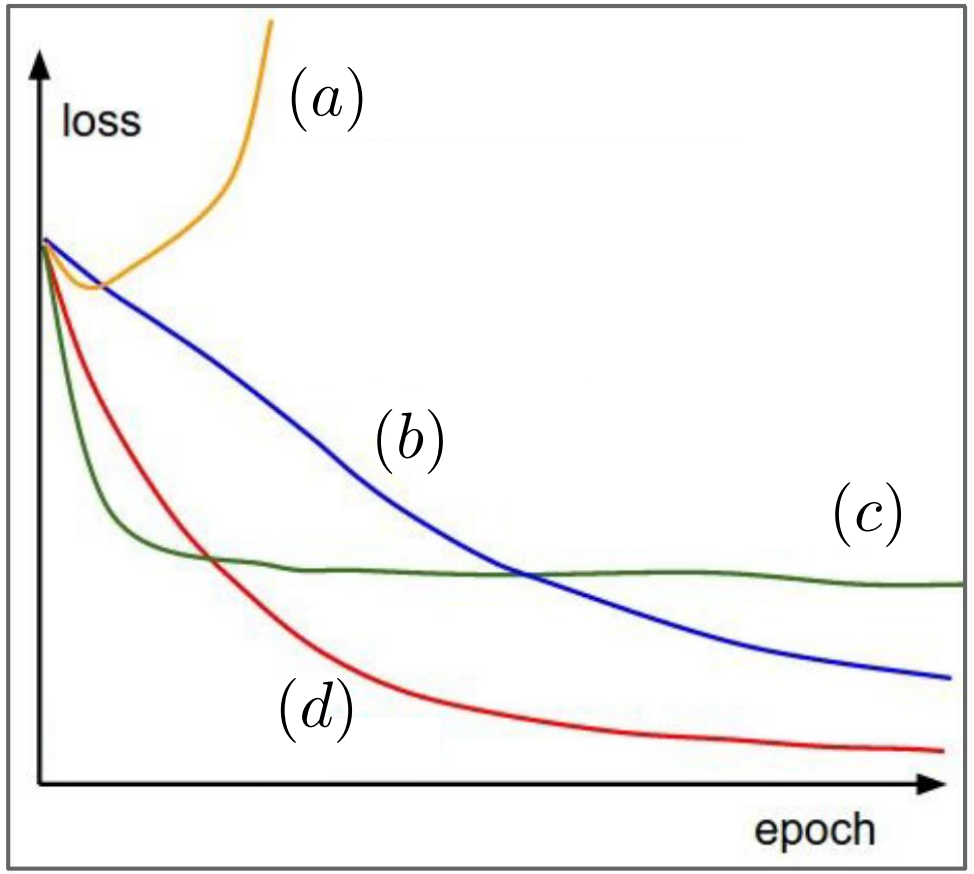
(4) Overfit a small portion of the data

100% accuracy on the training set (good)

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Slide: Andrej Karpathy

Babysitting Training

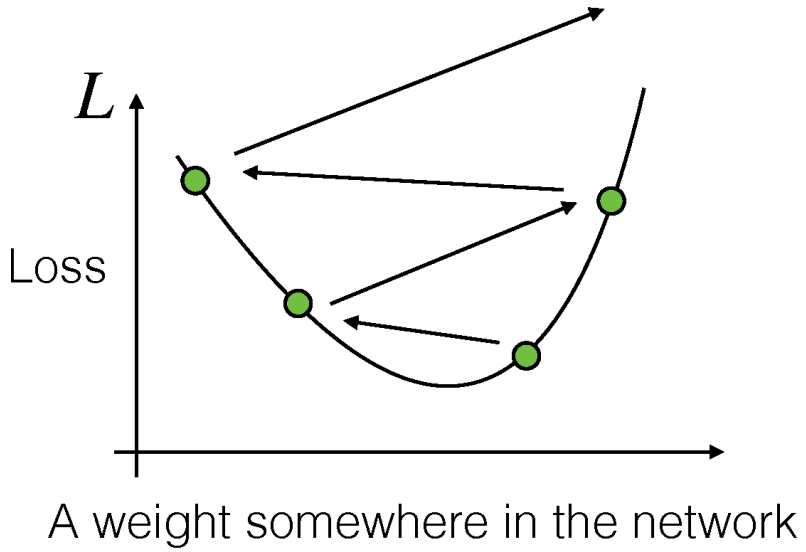


Q. Which learning curve is the best?

Q. How about learning rate?
Sort them in decreasing order.

Too large learning rate

Learning rate: 1e6 — what could go wrong?



High learning rate

Learning rate: 3e-3

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of class
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

Loss is inf —> still too high

But now we know we should be searching the range [1e-5 … 1e-3]

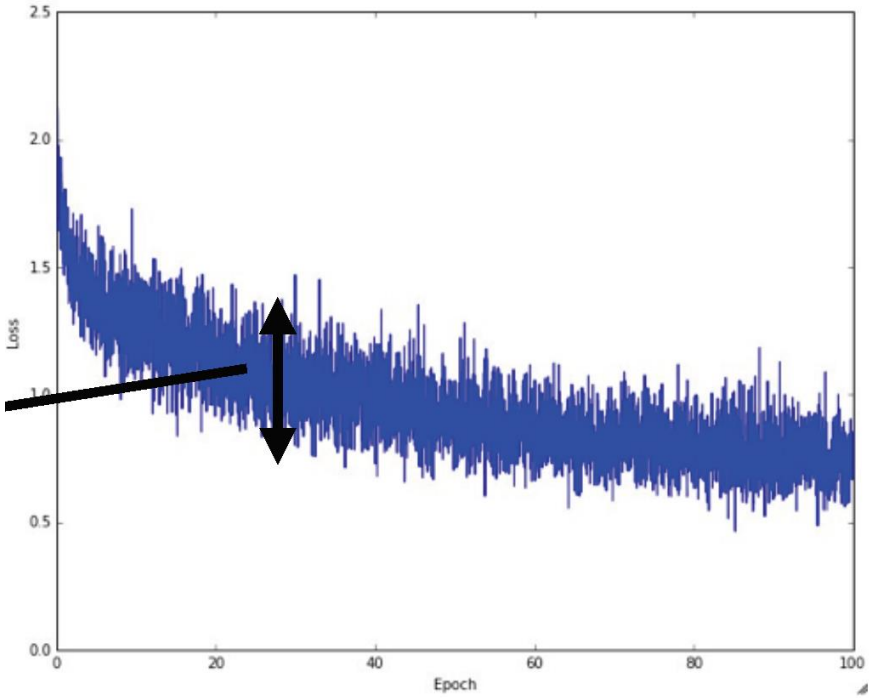
Slide: Andrej Karpath

Find a learning rate

- Grid search
- Coarse to fine search
  - Training models for a few epochs (coarse)
  - Once a reasonable learning rate is found, then train longer running time (fine)

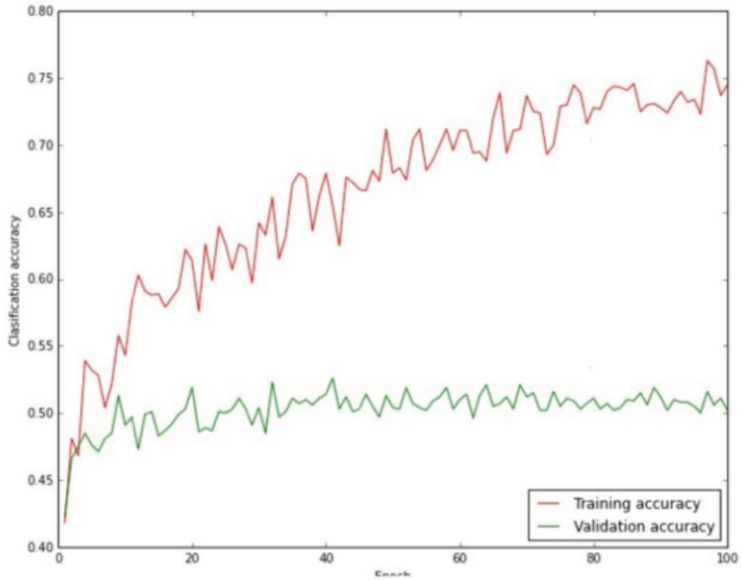
Problems & Solutions

- What if loss does not change much?
  - Too small learning rate
  - or too strong regularization
- What if loss oscillates a lot?
  - Increase the mini-batch size

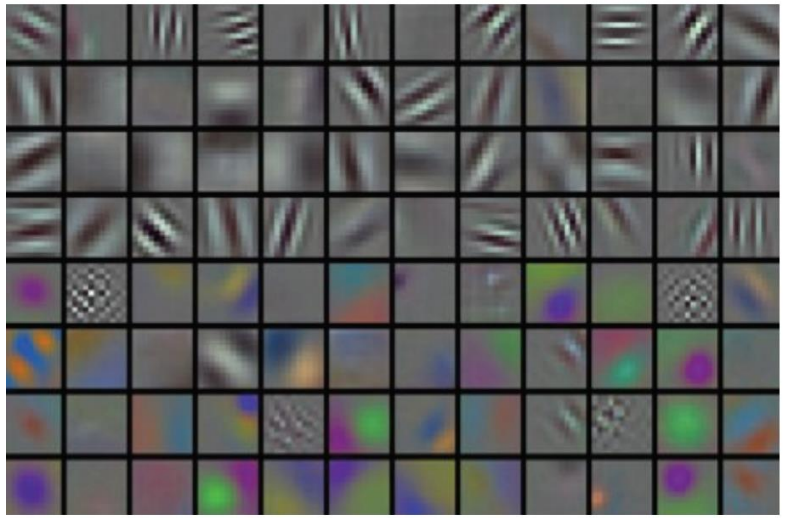


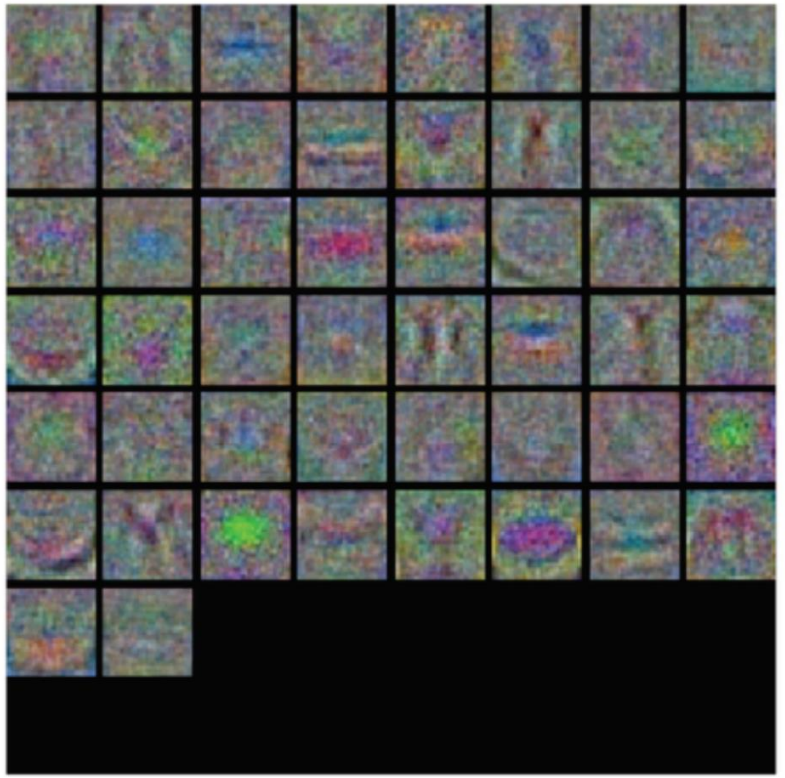
Overfitting vs. Underfitting

- Is this overfitting or underfitting?
  - Overfitting has a big gap: high training accuracy but low validation/test accuracy
  - Underfitting has a small/no gap between training/test accuracy and both are low.



Monitoring training via visualization of weights





Learning rate scheduler

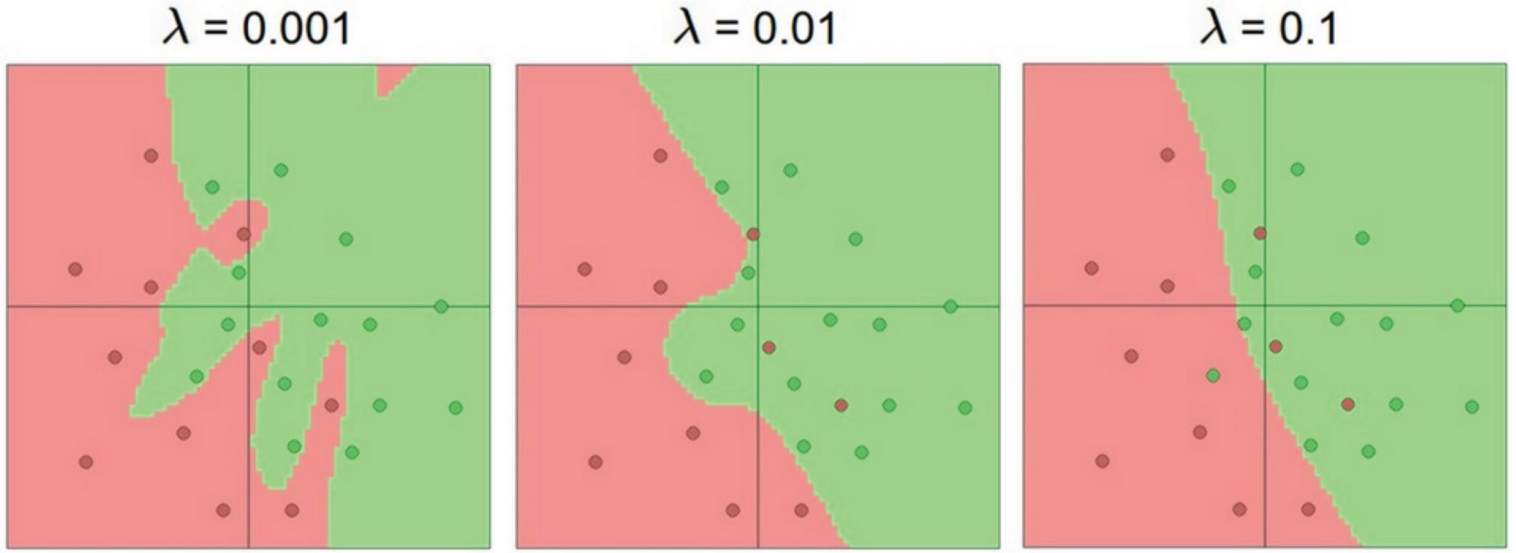
- Decrease LR by a factor of r every epoch (or every N iters)
- Use some equations sqrt(1-t/max_t)
- 1/t
- LR_{t+1} = LR_t exp(-t)
- …

Regularization

Regularization

Regularization reduces overfitting:

$$L = L_{\text{data}} + L_{\text{reg}} \qquad L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2$$



[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]

Example Regularizers

L2 regularization $L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2$

(L2 regularization encourages small weights)

L1 regularization $L_{\text{reg}} = \lambda ||W||_1 = \lambda \sum_{ij} |W_{ij}|$

(L1 regularization encourages sparse weights:
weights are encouraged to reduce to exactly zero)

"Elastic net" $L_{\text{reg}} = \lambda_1 ||W||_1 + \lambda_2 ||W||_2^2$

(combine L1 and L2 regularization)

Max norm

Clamp weights to some max norm $||W||_2^2 \leq c$

Weight decaying is equivalent to Regularization

Regularization is also called "weight decay" because the weights "decay" each iteration:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \longrightarrow \frac{\partial L}{\partial W} = \lambda W$$

Gradient descent step:

$$W \leftarrow W - \alpha\lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

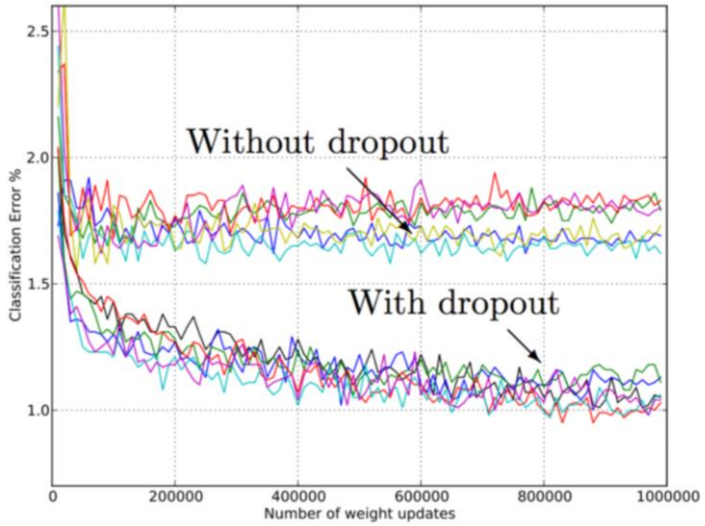
Weight decay: $\alpha\lambda$ (weights always decay by this amount)

Note: biases are sometimes excluded from regularization

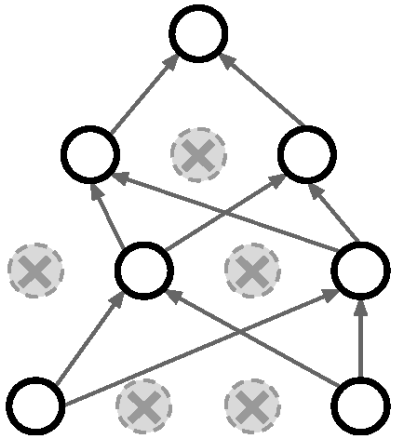
[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]

Dropout

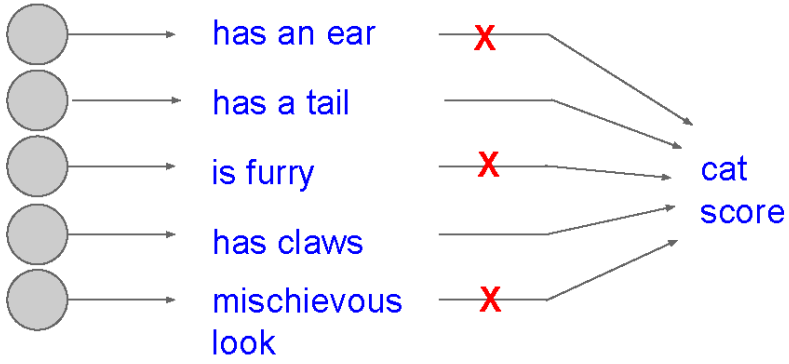
- Regularization
- Feature-level Data augmentation



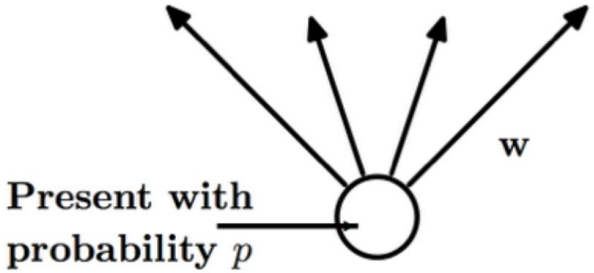
Srivasta et al., Dropout: A simple Way to Prevent Neural Networks from Overfitting, JMLR 2014



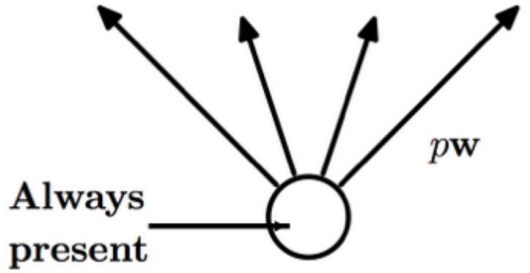
Forces the network to have a redundant representation;
Prevents co-adaptation of features



Interpretation: an ensemble of exponentially many models



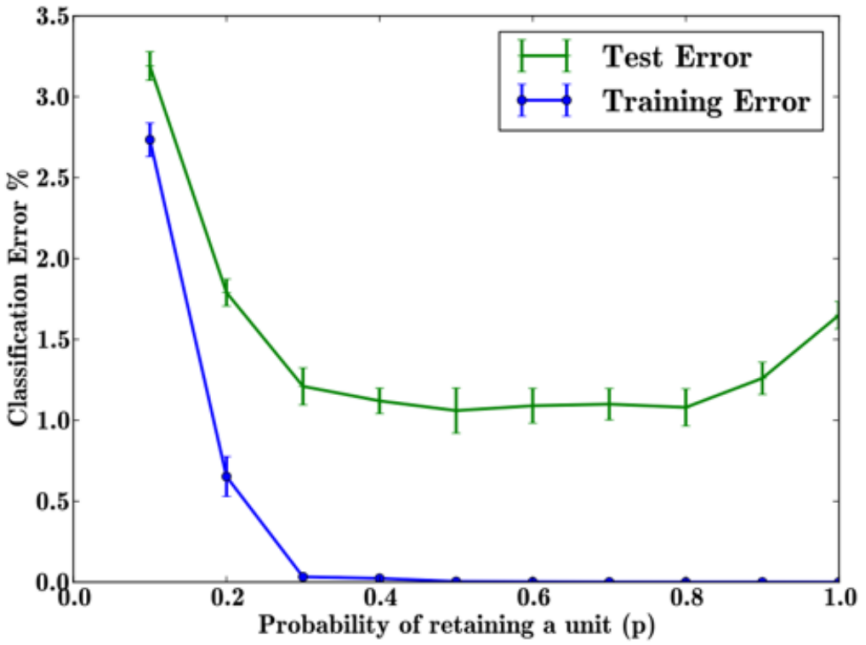
(a) At training time



(b) At test time

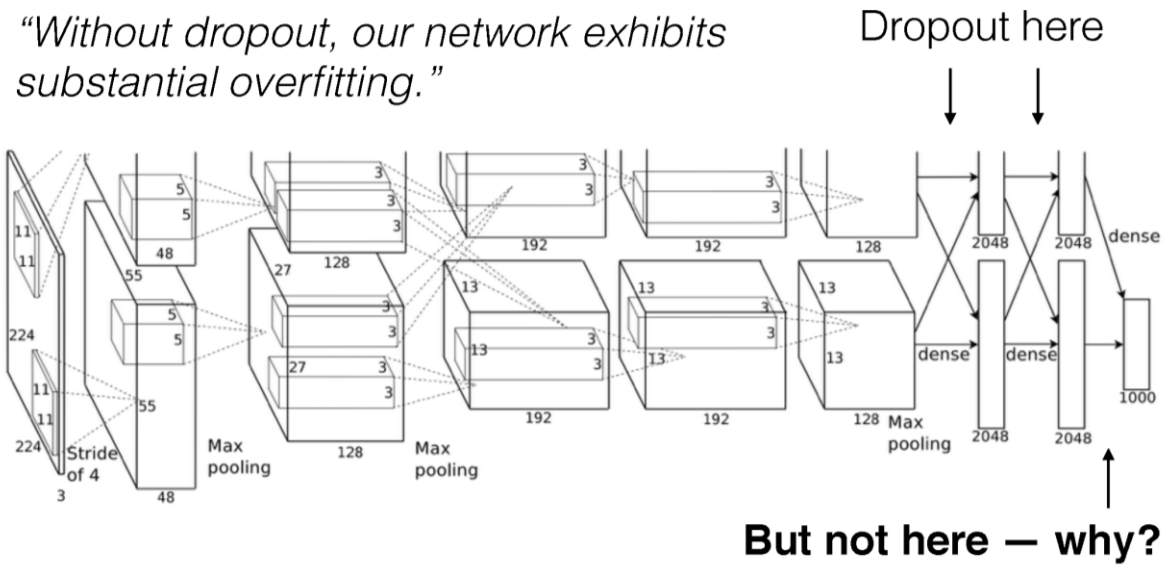
Dropout

How much dropout? Around p = 0.5



(a) Keeping n fixed.

"Without dropout, our network exhibits substantial overfitting."



[Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012]

[Srivasta et al, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014]

Dropout

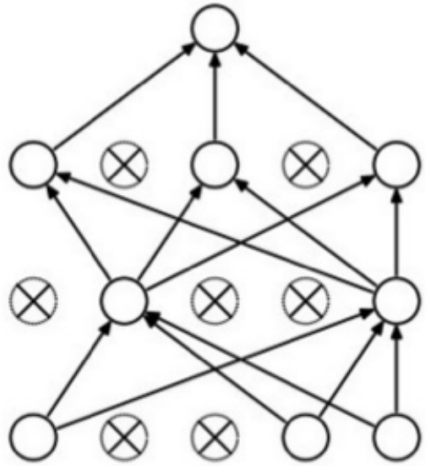
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # 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

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



(note, here X is a single input)

Figure: Andrej Karpathy

Dropout

Test time: scale the activations

Expected value of a neuron *h* with dropout:

$$E[h] = ph + (1-p)0 = ph$$

```
def predict(X):
  # 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
```

We want to keep the same expected value

Figure: Andrej Karpathy

Summary

- Preprocess the data (subtract mean)
  - Data Augmentation
- Initialize weights carefully
- Regularization
  - Norm, Dropout, Batch/group normalization
- SGD + Momentum
- Fine-tune from ImageNet
- Babysit the network as it trains

Questions?