2019
怪兽
学堂

CNN tricks

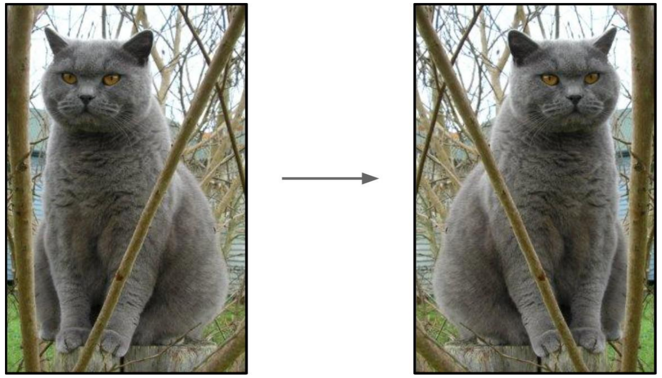
虾米

2019-03

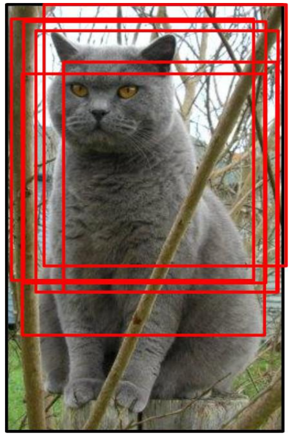
Outline

- ☑ Data augmentation
- ☑ Pre-processing
- ☑ Initializations
- ☑ During training
- ☑ Activation functions
- ☑ Regularizations
- ☑ Insights from figures
- ☑ Ensemble

Data augmentation



Flip horizontally



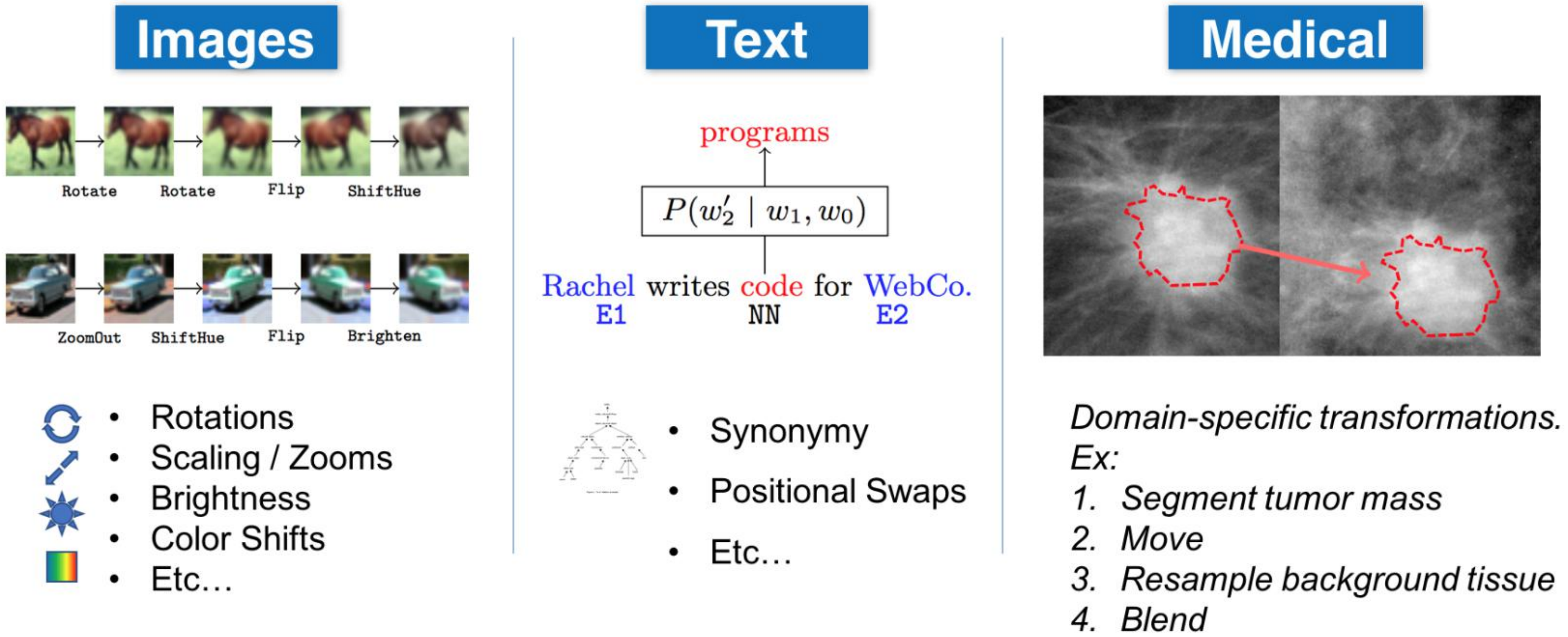
Random crops/scales

Color jittering

Random mix/combinations of :
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data augmentation

- **Learning to Compose Domain-Specific Transformations for Data Augmentation**

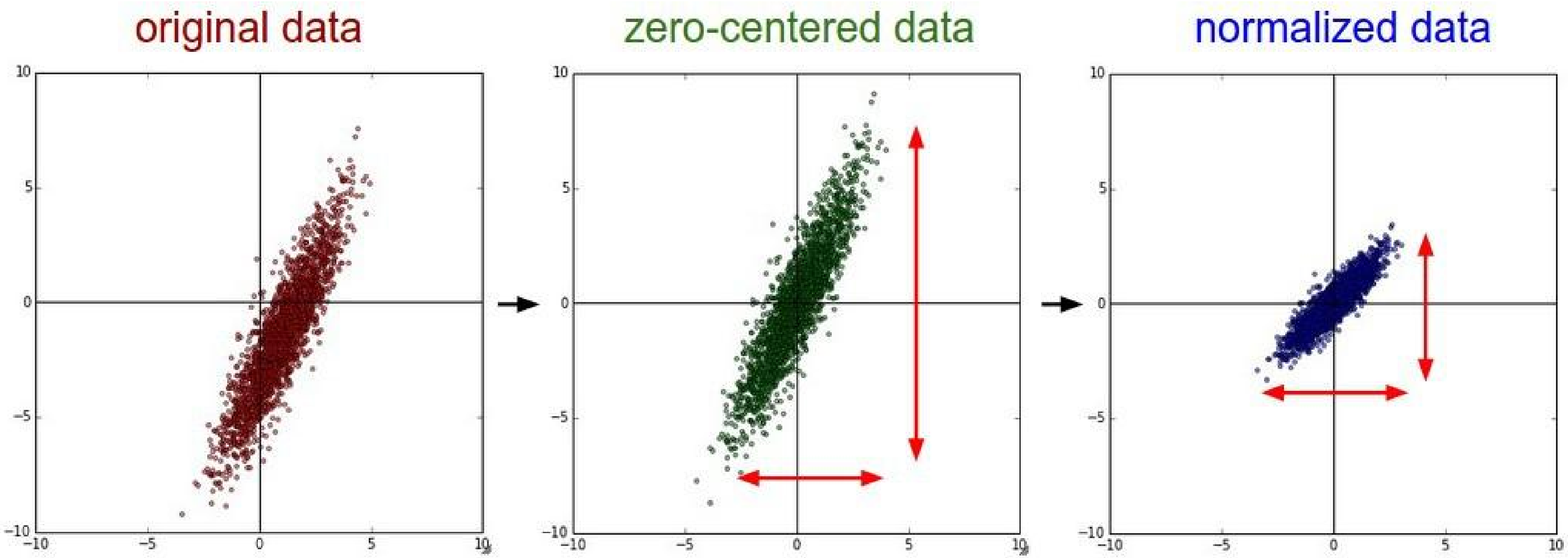


Data augmentation

- Be careful:

Rotate can not be used, Data augmentation should not change the object entity

Pre-processing



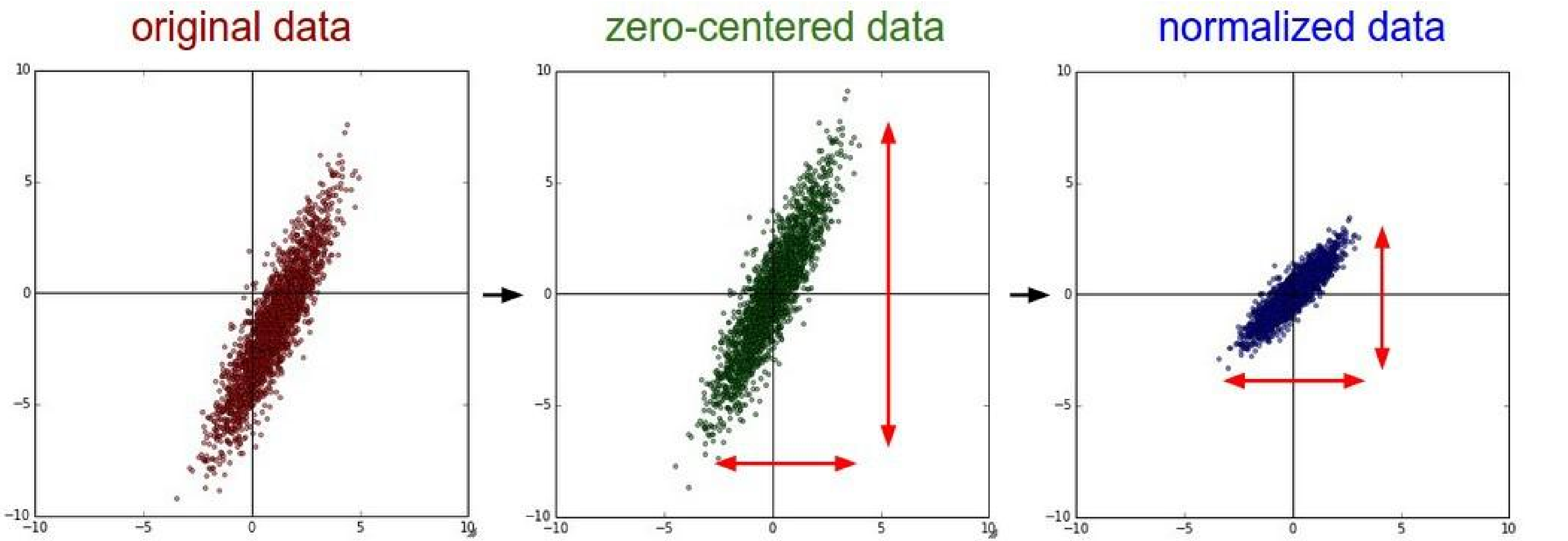

```
X -= np.mean(X, axis = 0)
```

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

Another way to normalize data is making the min and max along the dimension be -1 and 1, respectively.

Figures courtesy of Stanford course CS231n.

Pre-processing




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

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

It is **not** strictly necessary to perform this additional preprocessing step for the case of **IMAGE**.

Figures courtesy of Stanford course CS231n.

Initialization

- With proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative.

How about **all zero initialization**?

Initialization

- With proper data normalization it is reasonable to assume that approximately half of the weights will be positive and half of them will be negative.

How about **all zero initialization**?

They will also all compute the same gradients during backpropagation and undergo the exact same parameter updates.

Small random numbers

- We still want the weights to be very close to 0.

$$weights \sim 0.001 \times \underline{N(0,1)}$$

Uniform distribution is also ok.

Warning! Small numbers will diminish the "gradient signal" flowing backward through a network.

Calibrating the variances

The distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs.

```
w = np.random.randn(n) / sqrt(n)
```

This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

Current recommendation

They reached the conclusion that the variance of neurons in the network should be **2.0/n**.

```
w = np.random.randn(n) * sqrt(2.0/n)
```

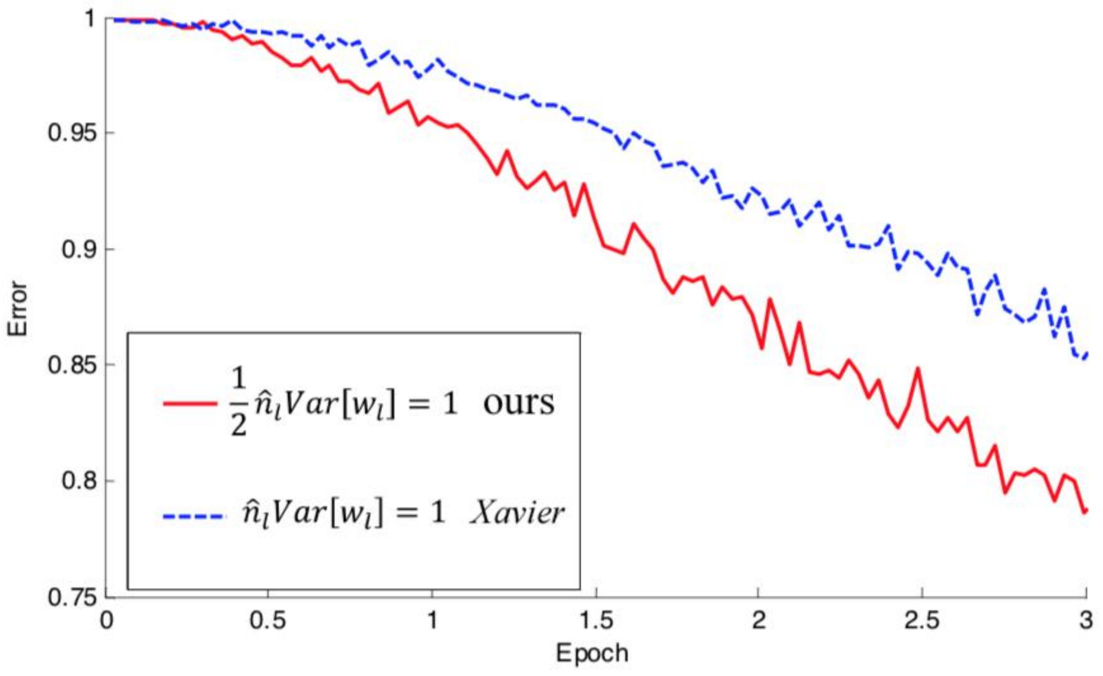


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and "*Xavier*" (blue)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

During training

Eliminate sizing headaches TIPS/TRICKS:

- start with image that has power-of-2 size
- for **conv layers**, use stride 1 filter size 3*3 pad input with a border of zeros (1 spatially)

This makes it so that: [W1, H1, **D1**] -> [W1, H1, **D2**] (i.e., spatial size exactly preserved)

- for **pool layers**, use pool size 2*2 (more = worse)

Learning

Gradient normalization:

- Divide the gradients by minibatch size. You won't need to change the learning rates (not too much, anyway), if you double the minibatch size (or halve it).

Learning rate (LR) schedule:

- A typical value of the LR is **0.1**;
- Use **a validation set**;
- Practical suggestion: if you see that you stopped making progress on the validation set, divide the LR by 2 (or by 5), and keep going.

Fine-tune

	very similar dataset	very different dataset
very little data	Use linear classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a large number of layers

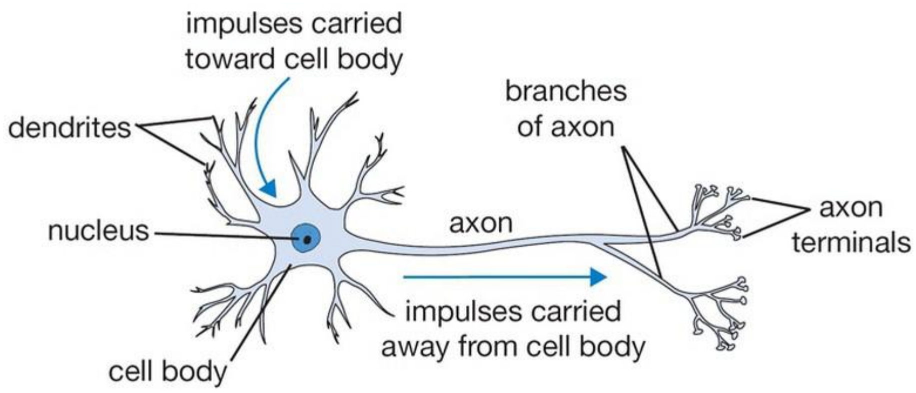
randomly-initialized

A smaller learning rate

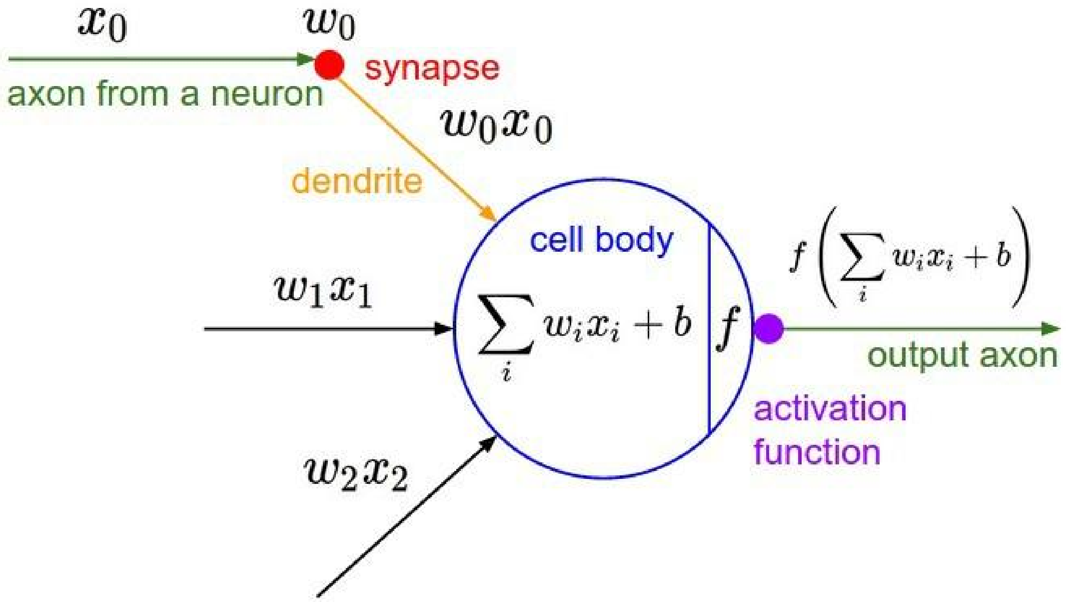
Activation functions

- Sigmoid
- tanh
- ReLU (Rectified Linear Unit)
- Leaky ReLU
- Parametric ReLU
- Randomized ReLU

Activation functions



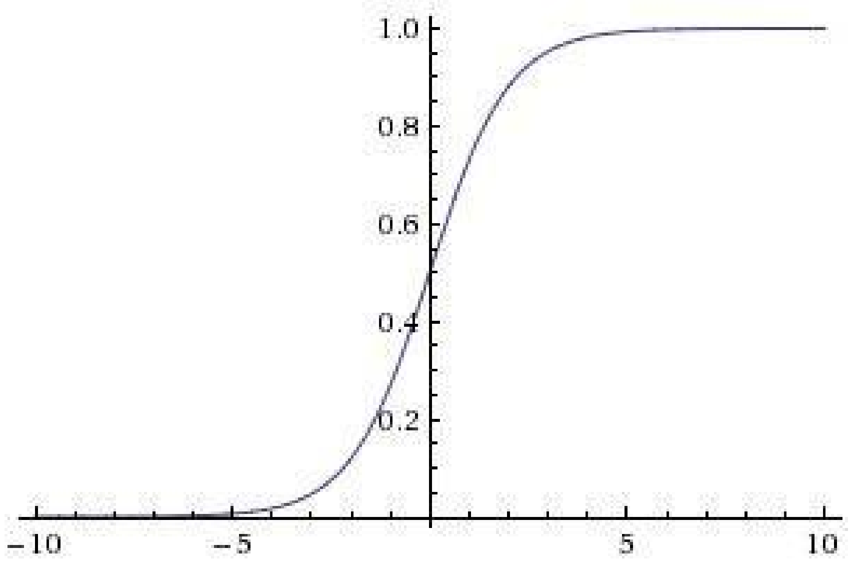
Non-linearity!



Figures courtesy of Stanford course of CS231n.

Activation functions

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



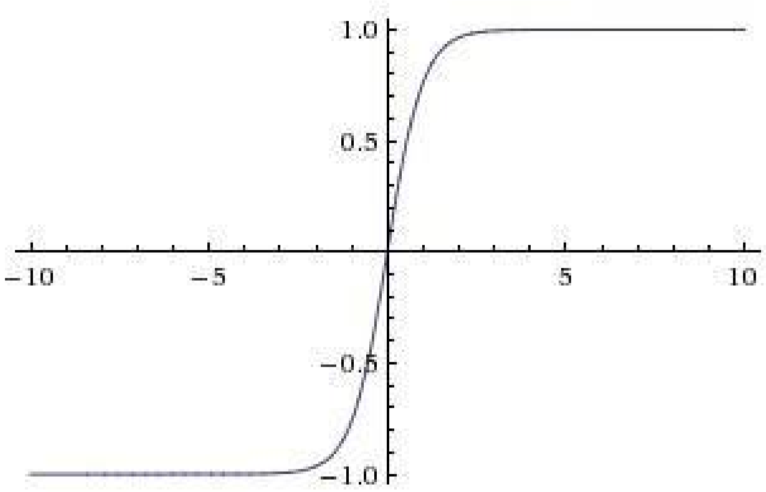
Sigmoid

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

2 BIG problems:

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

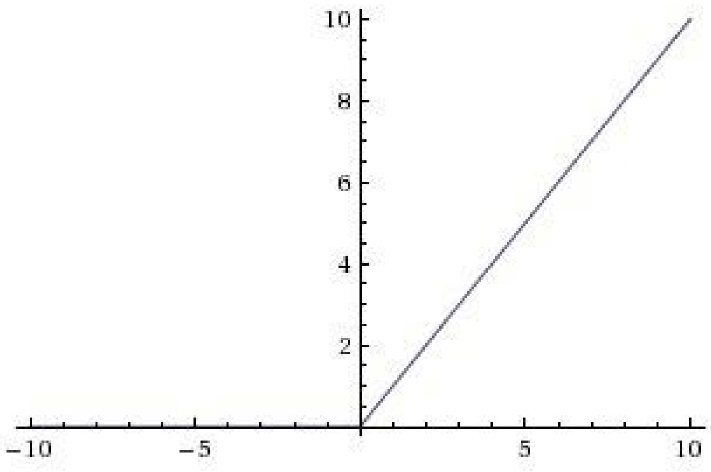
Activation functions



tanh(x)

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

Activation functions



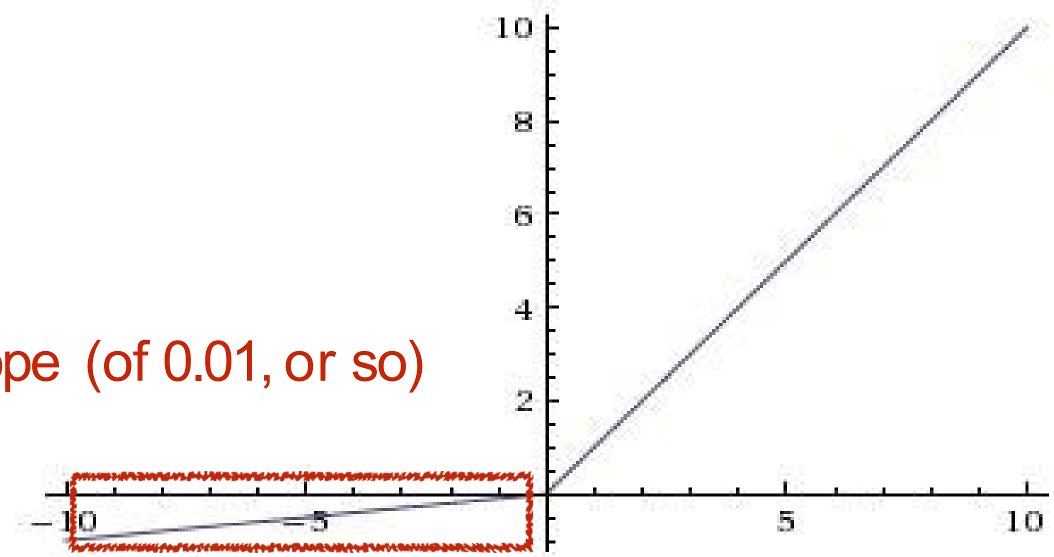
ReLU
(Rectified Linear Unit)

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

Just one annoying problem … what is the gradient when x<0?

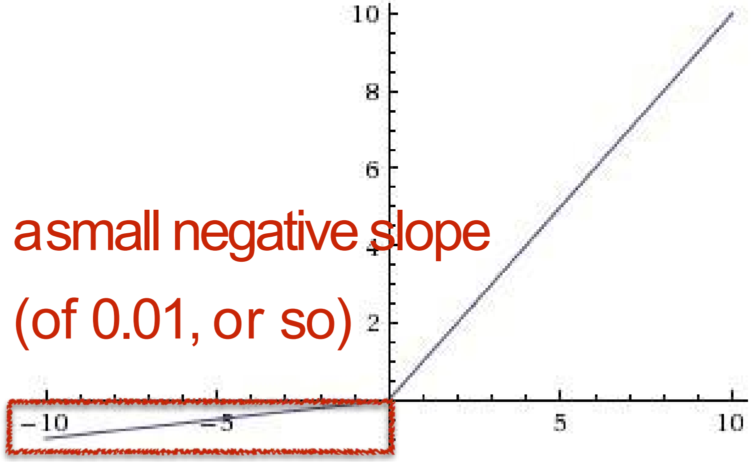
Activation functions

a small negative slope (of 0.01, or so)



Leaky ReLU

Activation functions



Leaky ReLU

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

Activation functions

Maxout "Neuron" (born in Jan.2013)

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters :(

Practical suggestions

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout
- Try out tanh but don't expect much
- Never use sigmoid

— Advised by F.F. Li and A. Karpathy

Some other trials

Variants of ReLU:

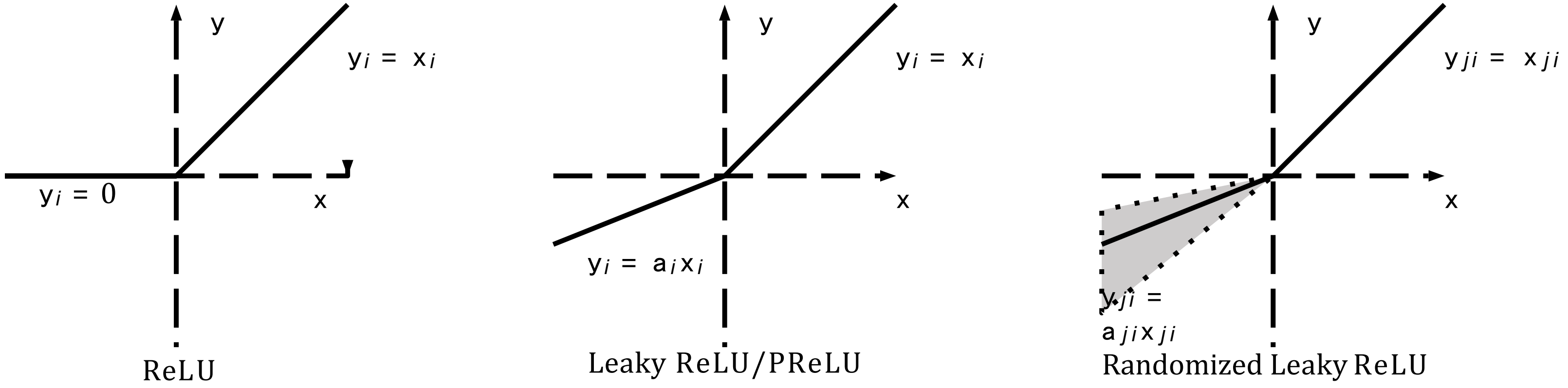


Figure 1: ReLU, Leaky ReLU, PReLU and RReLU. For PReLU, a_i is learned and for Leaky ReLU a_i is fixed. For RReLU, a_{ji} is a random variable keeps sam- pling in a given range, and remains fixed in testing.

[B. Xu, N.Wang,T. Chen and M. Li, arXiv: 1505.00853v1]

PReLU

Parametric ReLU:

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ \boxed{a_i} y_i, & \text{if } y_i \le 0 \end{cases}. \qquad (1)$$

is learned for the i-th channel

BP on PReLU:

Objective function

$$\frac{\partial \mathcal{E}}{\partial a_i} = \sum_{y_i} \frac{\partial \boxed{\mathcal{E}}}{\partial f(y_i)} \frac{\partial f(y_i)}{\partial a_i}, \qquad (2)$$

Updating \alpha — the momentum method:

$$\Delta a_i := \mu \Delta a_i + \epsilon \frac{\partial \mathcal{E}}{\partial a_i}.$$

RReLU

Randomized ReLU:

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}. \qquad (1)$$

$$a_i \sim 1/U(l, u).$$

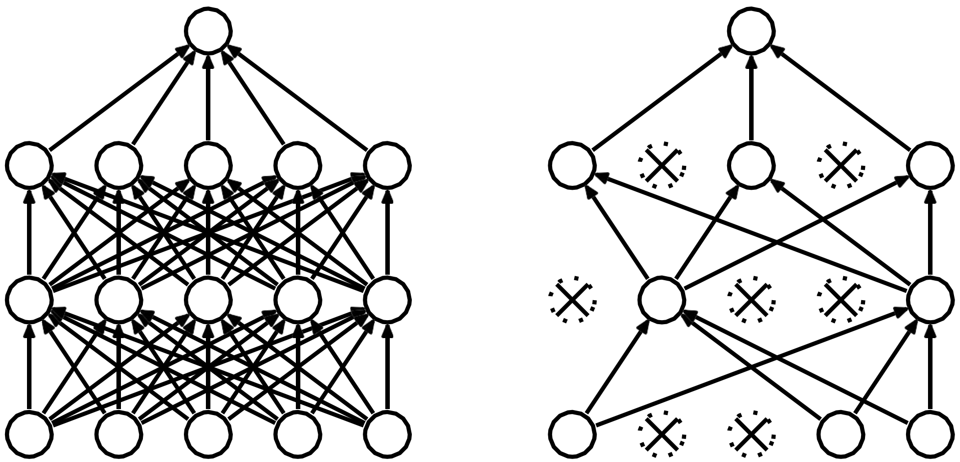
Test stage: only use the average "a" to get prediction results.

$$a_i = 2/(l + u) = \frac{1}{(l + u)/2}$$

Regularization

- L2 regularization
- L1 regularization
- L1 + L2 can also be combined
- Max norm constraints
- Dropout

$$\frac{1}{2}\lambda\omega^2 \qquad \lambda|\omega| \qquad ||\omega||_2 < c$$

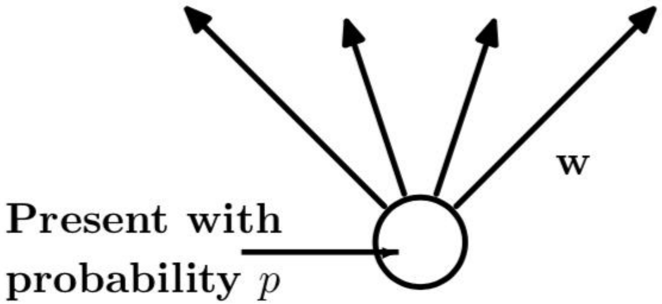


(a) Standard Neural Net (b) After applying dropout.

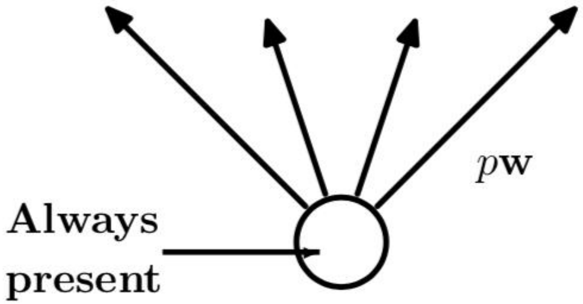
[N. Srivastava et al., JMLR'14]

Regularization

NOTICE!



(a) At training time



(b) At test time

Performance comparisons:

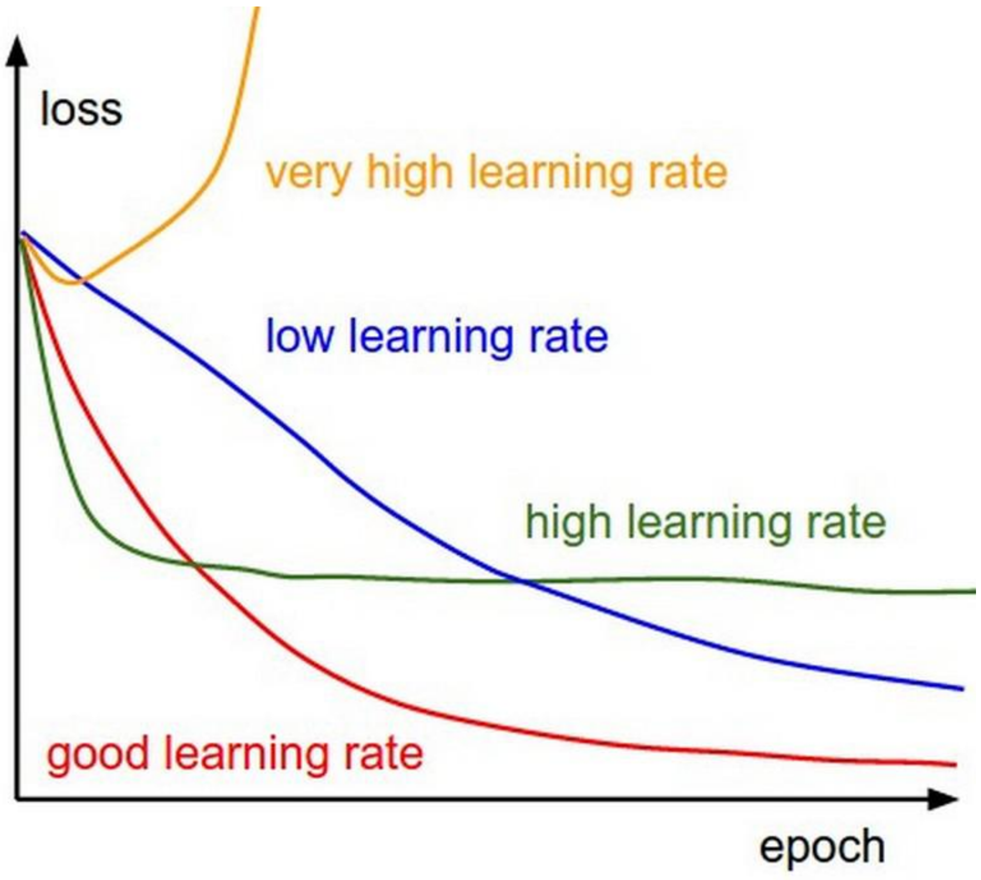
Method	Test Classiftcation error %
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

Table 9: Comparison of di↵erent regularization methods on MNIST.

[N. Srivastava et al., JMLR'14]

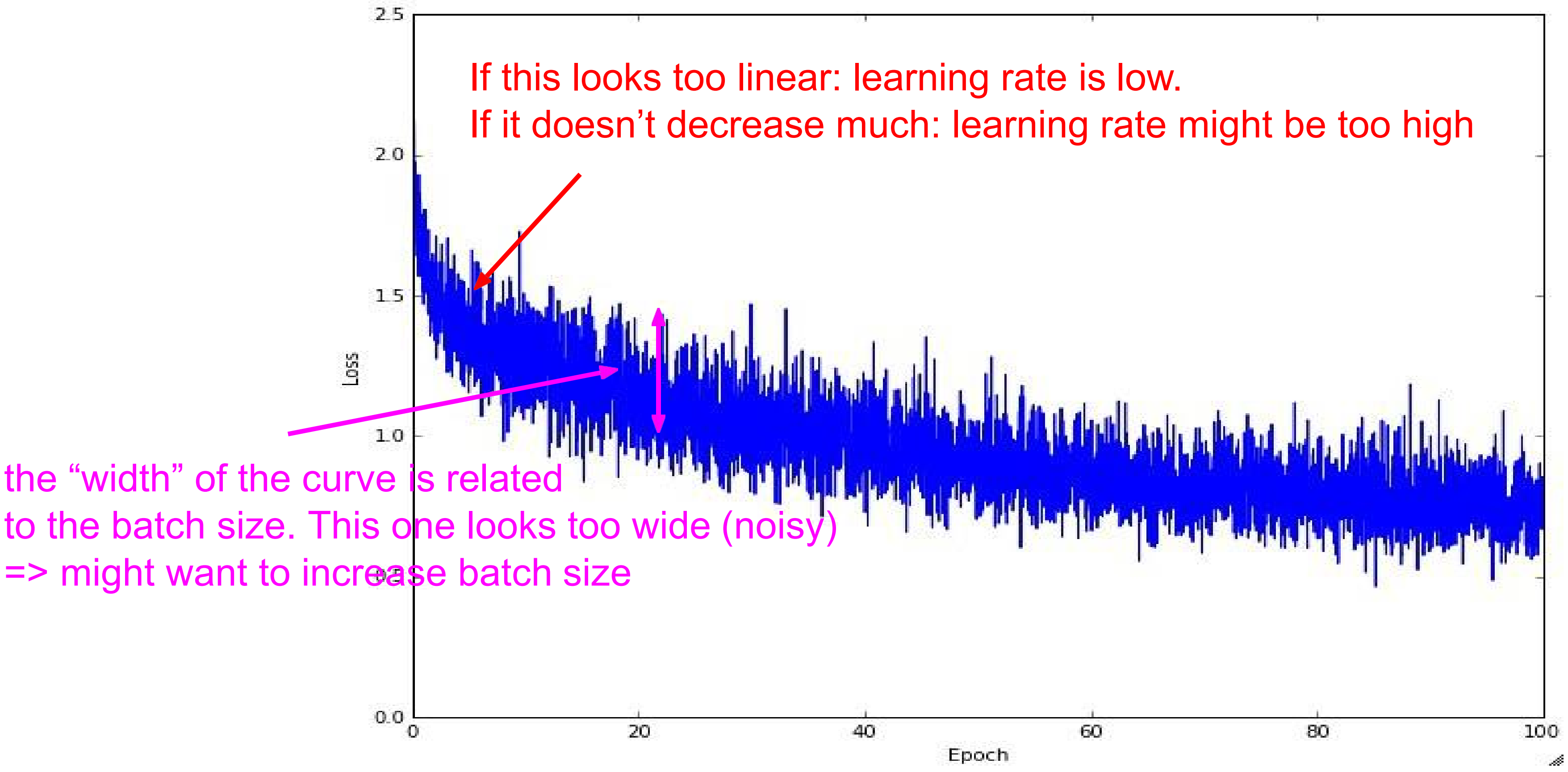
Insights from figures

The learning rate:



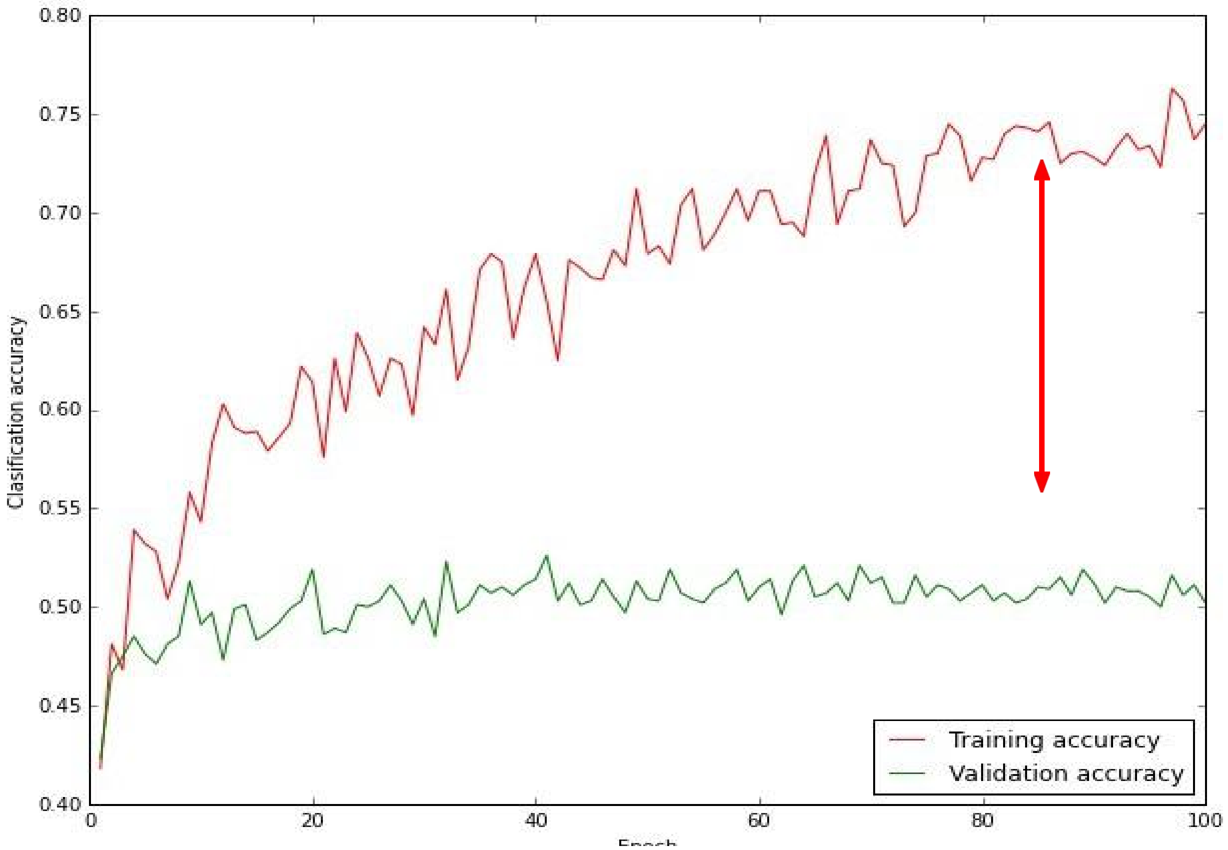
Insights from figures

The loss curve:



Insights from figures

The accuracy:



big gap = overfitting

=> increase regularization strength

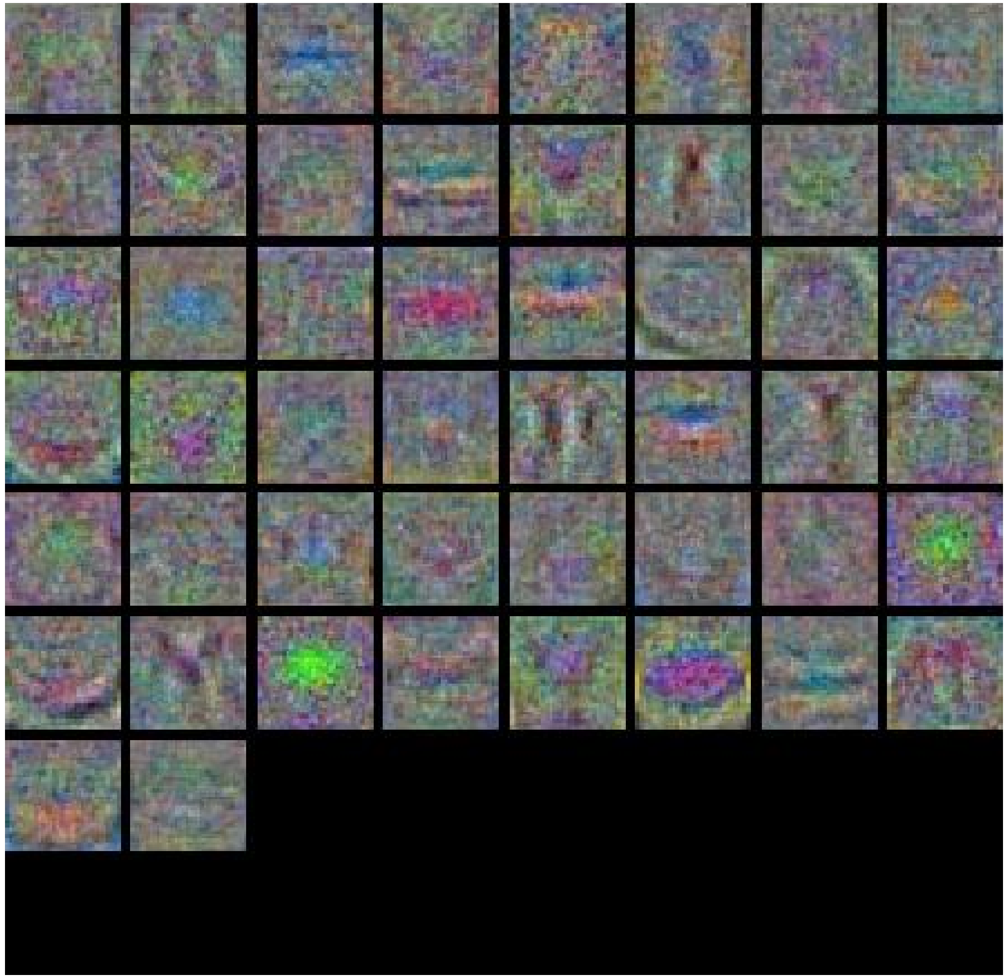
no gap

=> increase model capacity

Insights from figures

Visualizing first-layer weights:

Noisy weights => Regularization maybe not strong enough



Ensemble

- **Same model, different initializations**. Use cross-validation to determine the best hyperparameters, then train multiple models with the best set of hyperparameters but with different random initialization. The danger with this approach is that the variety is only due to initialization.
- **Top models discovered during cross-validation**. Use cross-validation to determine the best hyperparameters, then pick the top few (e.g. 10) models to form the ensemble. This improves the variety of the ensemble but has the danger of including suboptimal models. In practice, this can be easier to perform since it doesn't require additional retraining of models after cross-validation
- **Different checkpoints of a single model**. If training is very expensive, some people have had limited success in taking different checkpoints of a single network over time (for example after every epoch) and using those to form an ensemble. Clearly, this suffers from some lack of variety, but can still work reasonably well in practice. The advantage of this approach is that is very cheap.

2019
怪兽
学堂

THANKS