Neural network training: The basics and beyond

Slides from Lana Lazebnik

Outline

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive optimization methods: SGD with momentum, RMSProp, Adam
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization
- Regularization
- Test time: averaging predictions, ensembles

Mini-batch SGD

- Iterate over epochs
  - Group data into mini-batches of size b
    - Compute gradient of the loss for the mini-batch $(x_1, y_1), \ldots, (x_b, y_b)$:

$$\nabla \hat{L} = \frac{1}{b} \sum_{i=1}^{b} \nabla l(w, x_i, y_i)$$

    - Update parameters:

$$w \leftarrow w - \eta \nabla \hat{L}$$

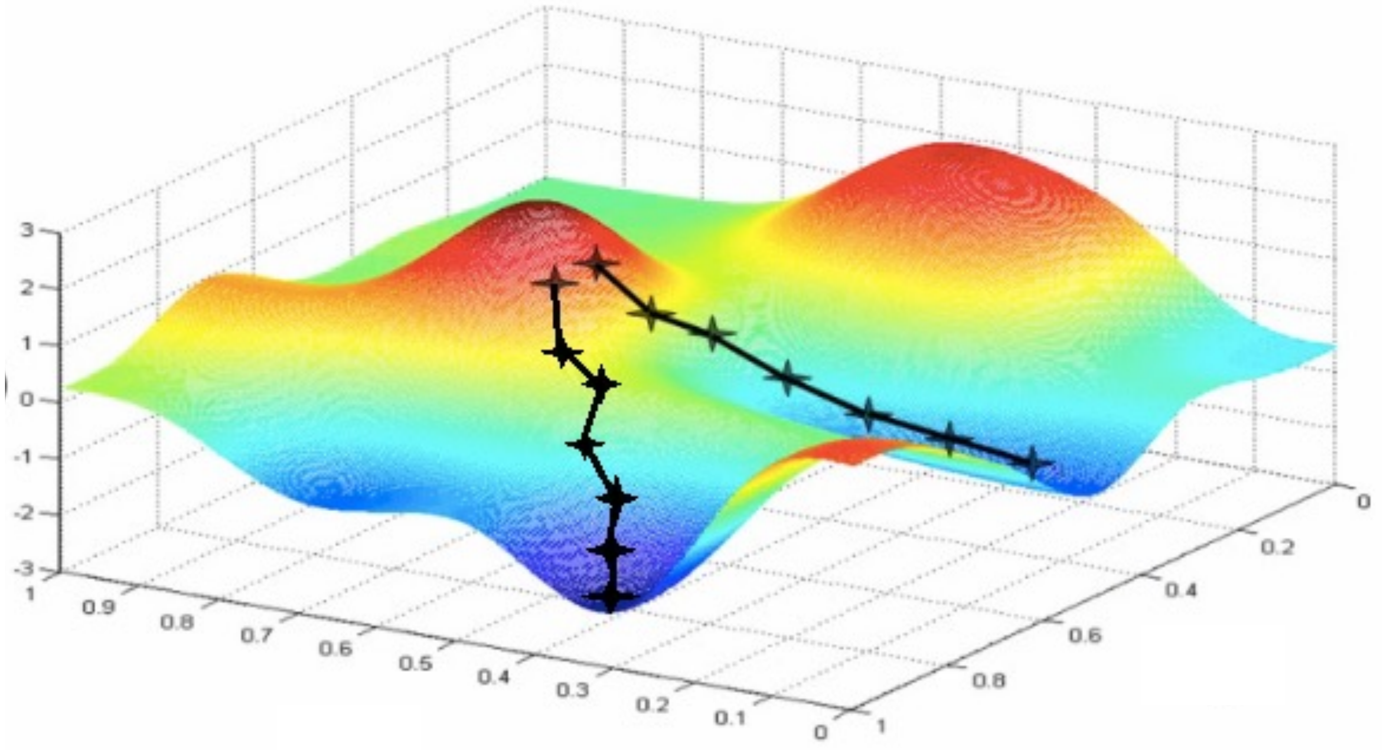
  - Check for convergence, decide whether to decay learning rate

- What are the hyperparameters?
  - Mini-batch size, learning rate decay schedule, deciding when to stop

Setting the mini-batch size

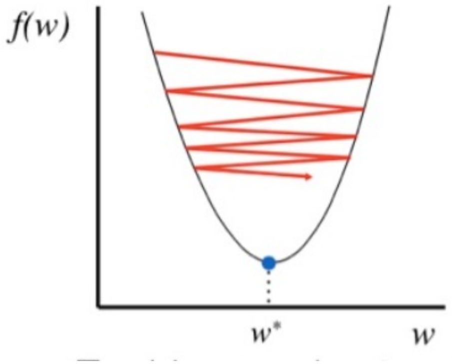
- Smaller mini-batches: less memory overhead, less parallelizable, more gradient noise (which could work as regularization – see, e.g., Keskar et al., 2017)
- Larger mini-batches: more expensive and less frequent updates, lower gradient variance, more parallelizable. Can be made to work well with good choices of learning rate and other aspects of optimization (Goyal et al., 2018)

Setting the learning rate

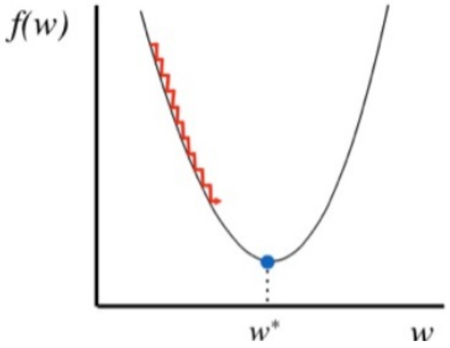


Setting the learning rate

Too high



Too low



Want: good *decay schedule*

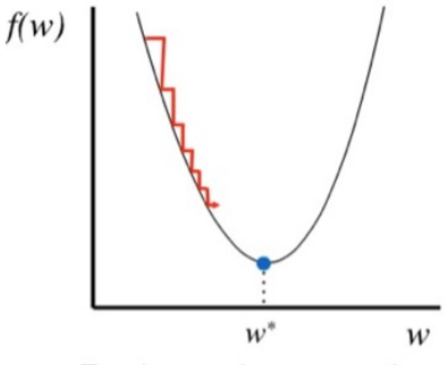
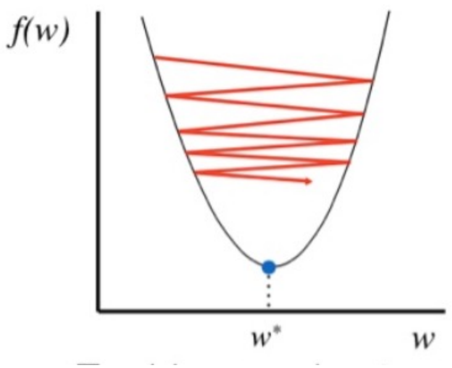


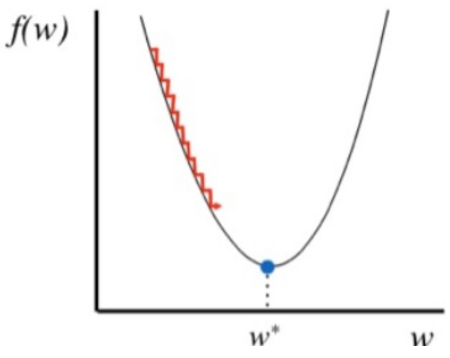
Figure source

Setting the learning rate

Too high



Too low



Want: good *decay schedule*

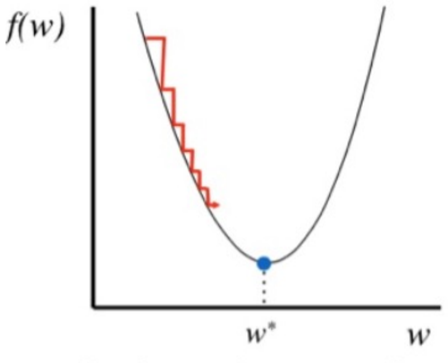
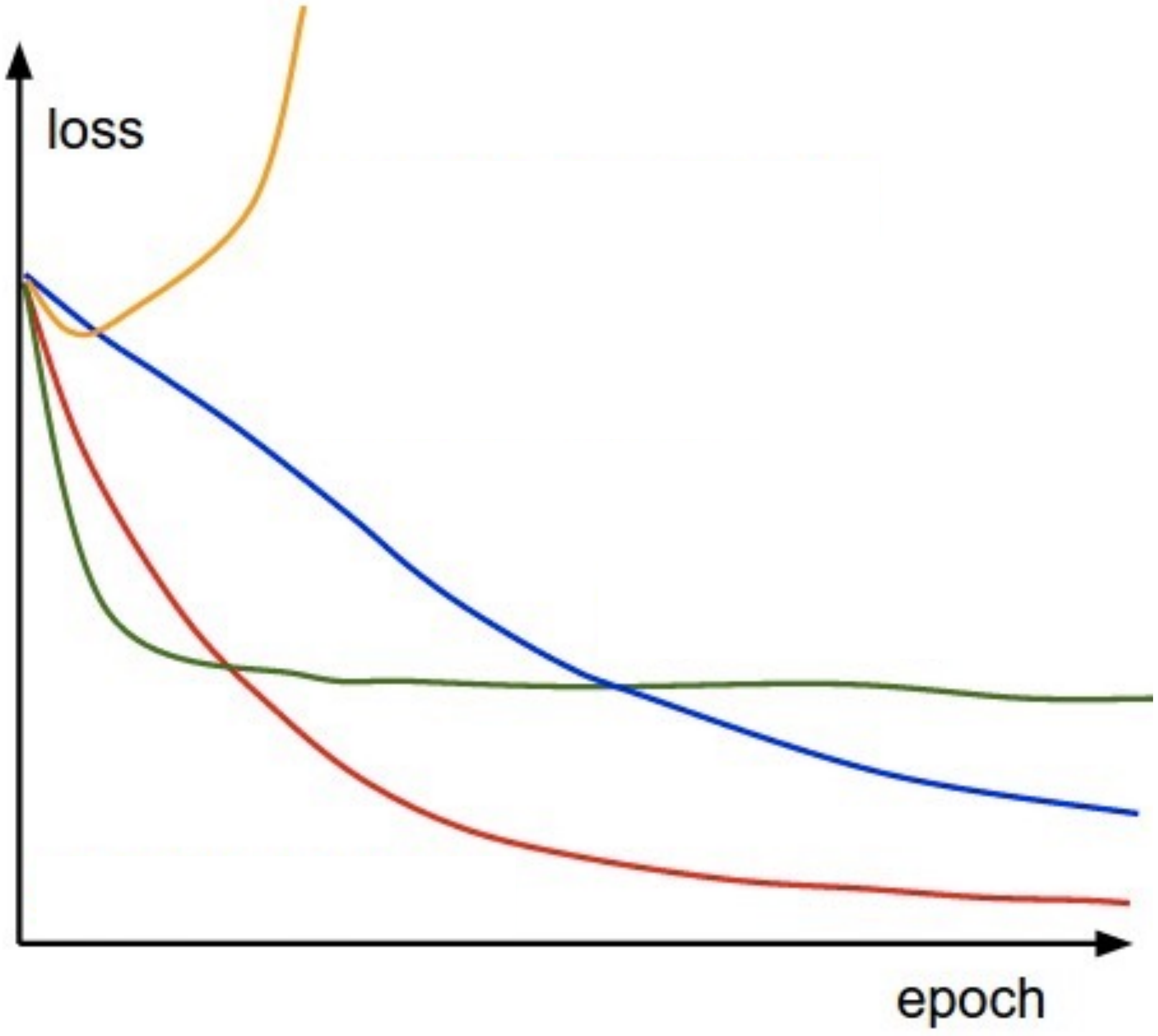


Figure source



Source: Stanford CS231n

Learning rate decay

- Decay formulas
  - Exponential: $\eta_t = \eta_0 e^{-kt}$, where η_0 and k are hyperparameters, t is the iteration or epoch number
  - Inverse: $\eta_t = \eta_0/(1 + kt)$
  - Inverse sqrt: $\eta_t = \eta_0/\sqrt{t}$
  - Linear: $\eta_t = \eta_0(1 - t/T)$, where *T* is the total number of epochs
  - Cosine: $\eta_t = \frac{1}{2}\eta_0(1 + \cos(t\pi/T))$

Learning rate decay

- Decay formulas
- Most common in practice:
  - **Step decay:** reduce rate by a constant factor every few epochs, e.g., by 0.5 every 5 epochs, 0.1 every 20 epochs
  - **Manual:** watch validation error and reduce learning rate whenever it stops improving
    - “Patience” hyperparameter: number of epochs without improvement before reducing learning rate

A typical phenomenon

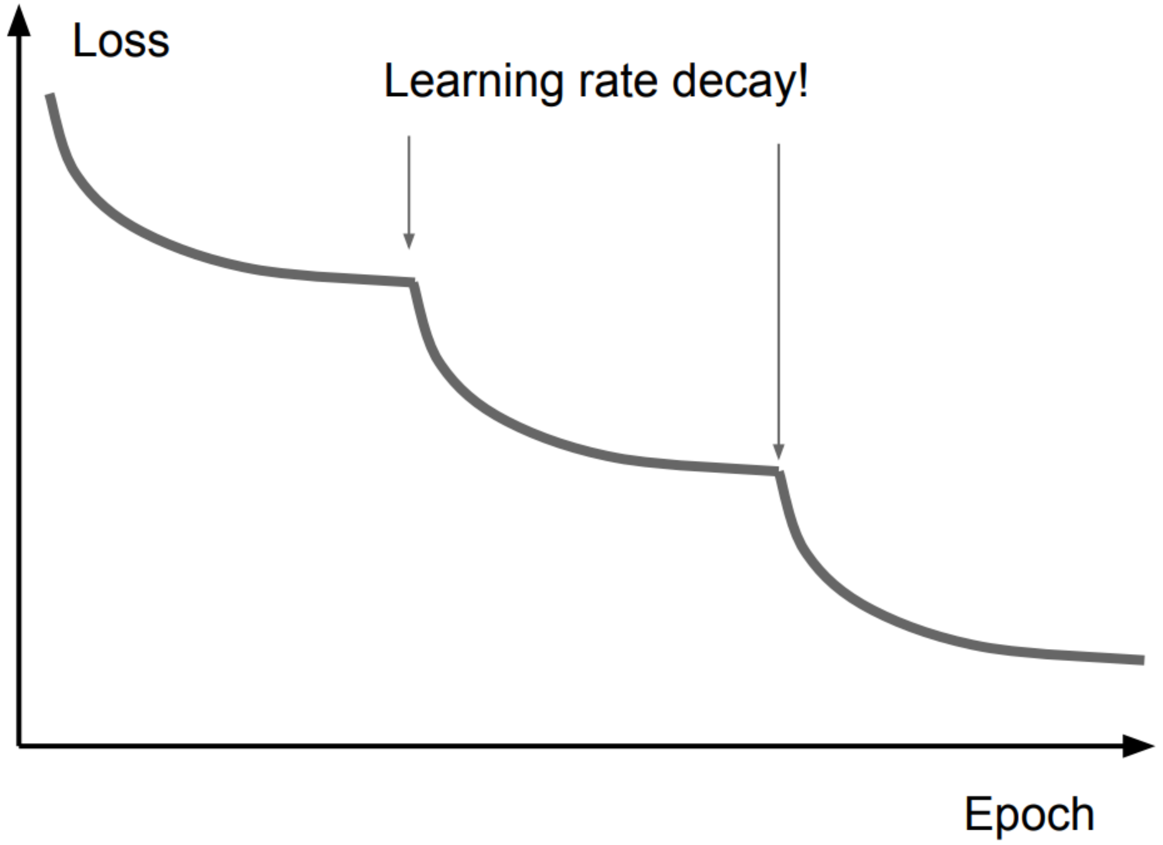


Image source: [Stanford CS231n]

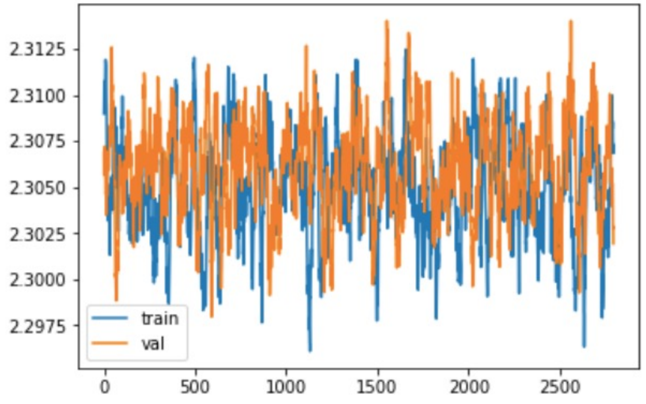
Possible explanation

[Image source]

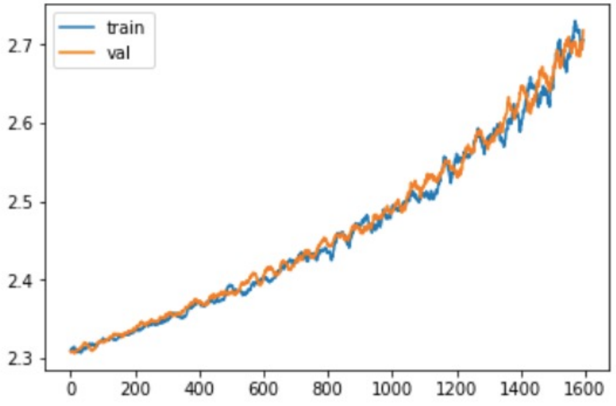
Learning rate decay

- Decay formulas
- Most common in practice:
  - **Step decay:** reduce rate by a constant factor every few epochs, e.g., by 0.5 every 5 epochs, 0.1 every 20 epochs
  - **Manual:** watch validation error and reduce learning rate whenever it stops improving
    - “Patience” hyperparameter: number of epochs without improvement before reducing learning rate
- **Warmup:** train with a low learning rate for a first few epochs, or linearly increase learning rate before transitioning to normal decay schedule (Goyal et al., 2018)

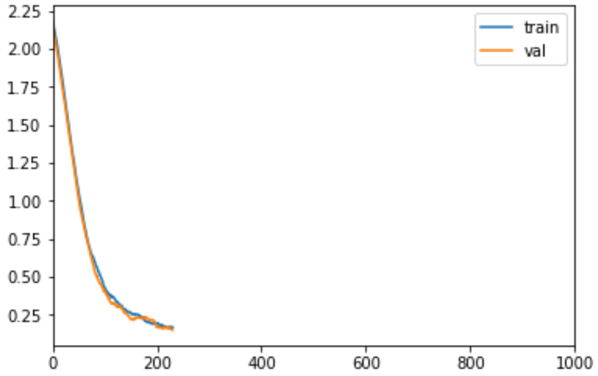
Diagnosing learning curves: Obvious problems



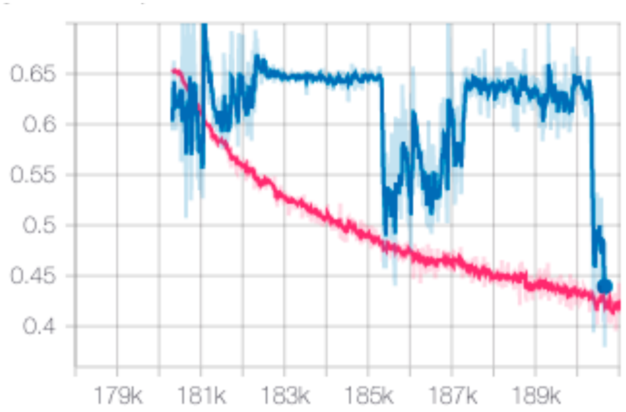
Not training
Bug in update calculation?



Error increasing
Bug in update calculation?



Get NaNs in the loss after a number of iterations:
Numerical instability



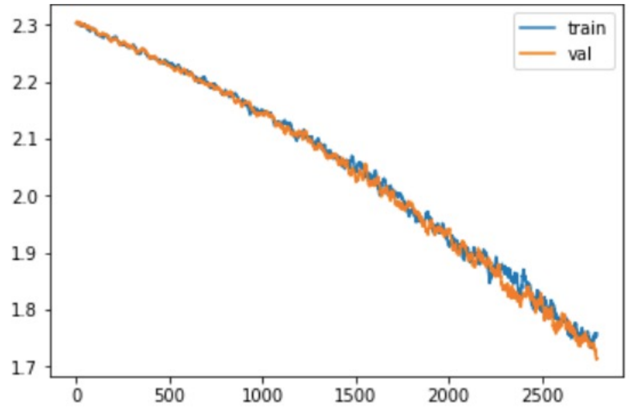
Shuffling off

Shuffling on

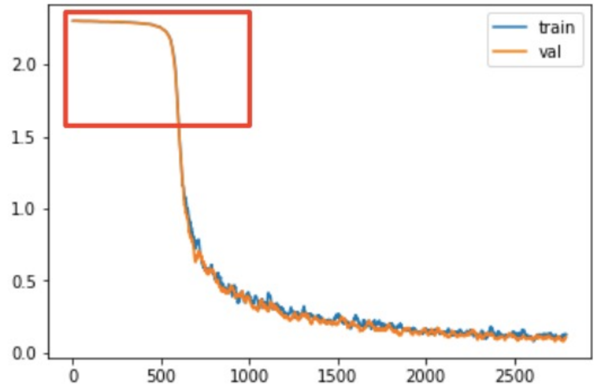
Weird cyclical patterns in loss:
Data not shuffled

Source: [Stanford CS231n]

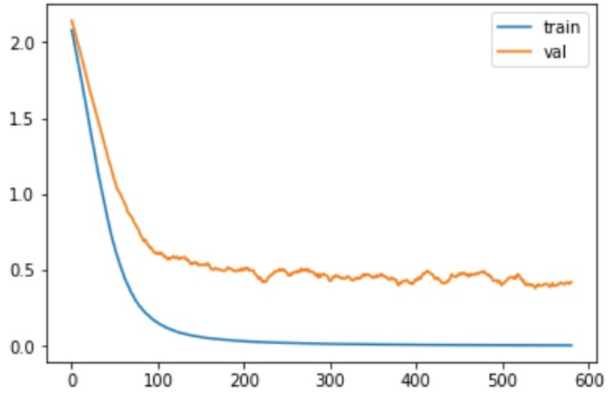
Diagnosing learning curves: Subtler behaviors



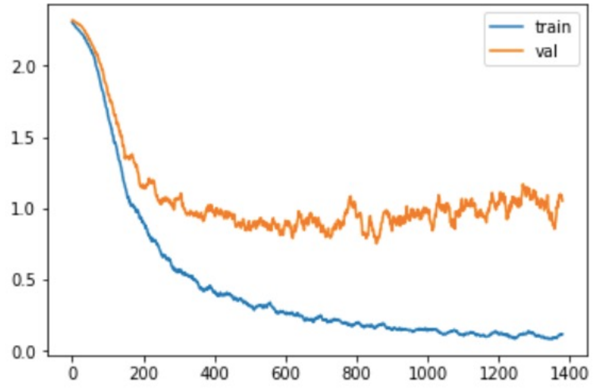
Not converged yet
Keep training, possibly increase learning rate



Slow start
Bad initialization?



Possible overfitting



Definite overfitting

Source: [Stanford CS231n]

When to stop training?

- Monitor validation error to decide when to stop
  - “Patience” hyperparameter: number of epochs without improvement before stopping
  - *Early stopping* can be viewed as a kind of regularization

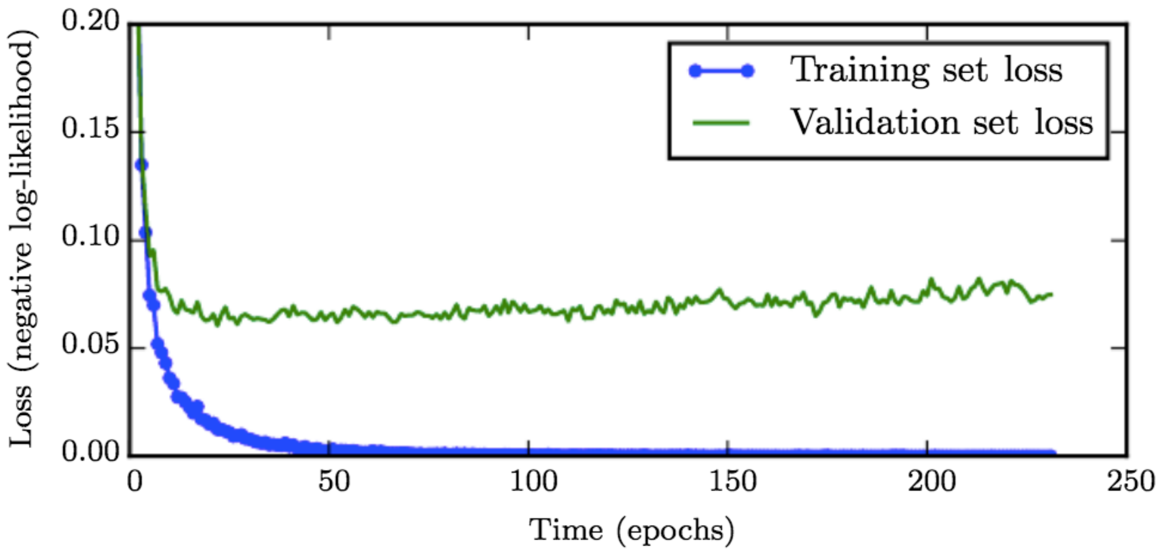
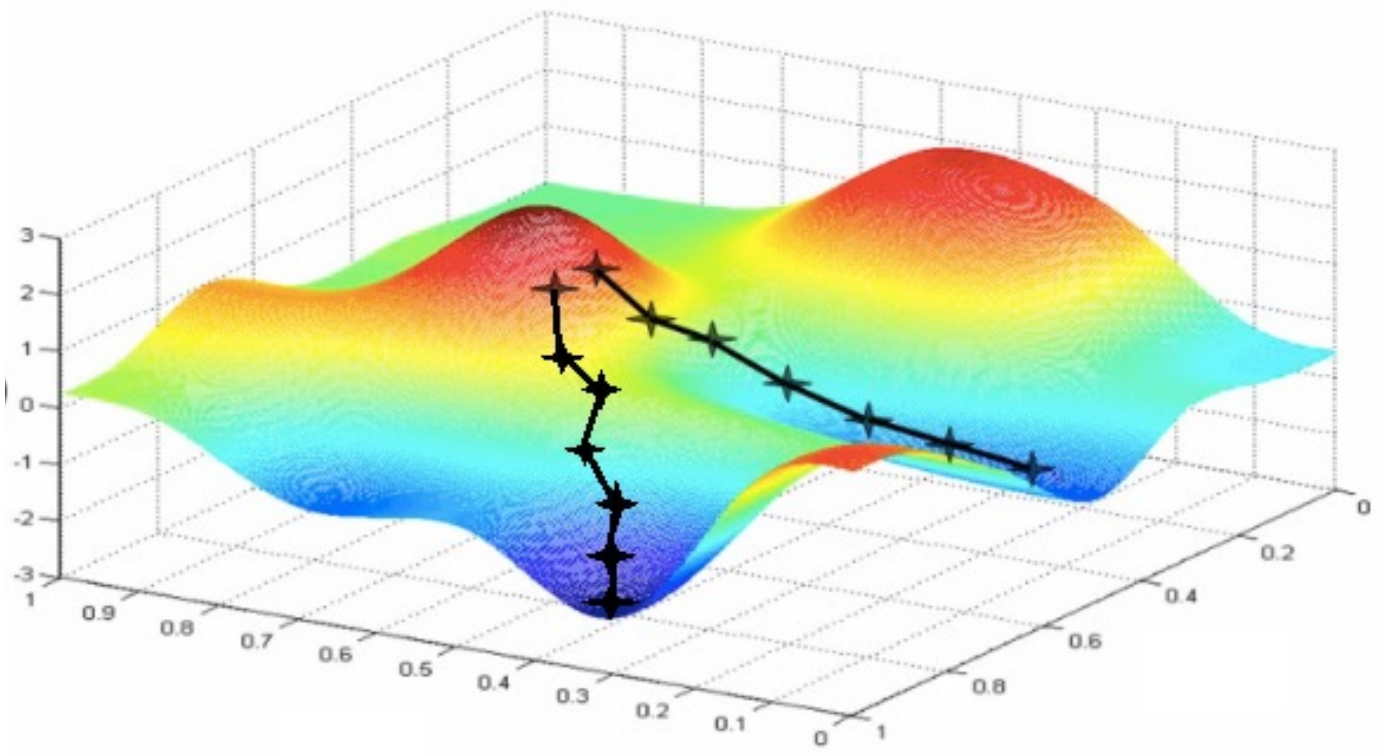


Figure from Deep Learning Book

Neural network training: The basics and beyond

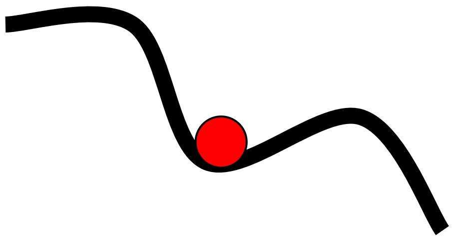
- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive optimization methods: SGD with momentum, RMSProp, Adam
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization
- Regularization
- Test time: averaging predictions, ensembles

Where does SGD run into trouble?

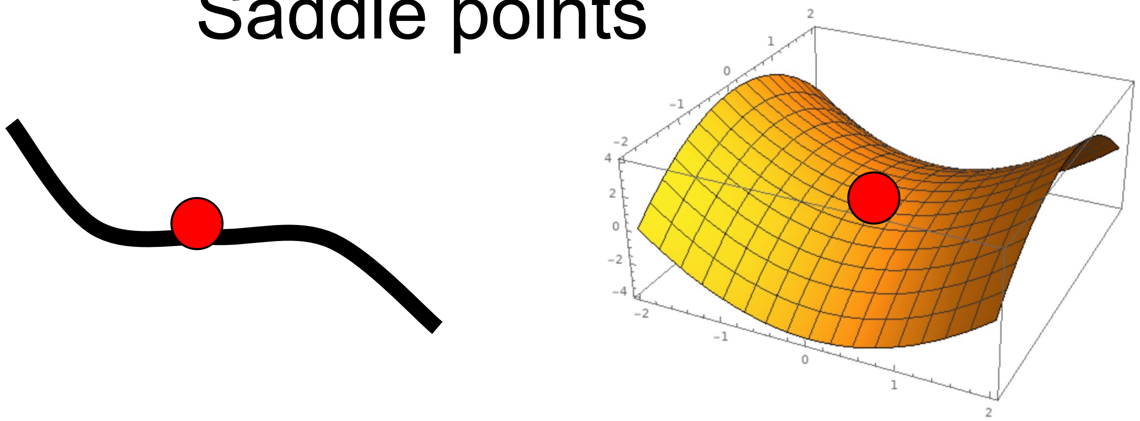


Where does SGD run into trouble?

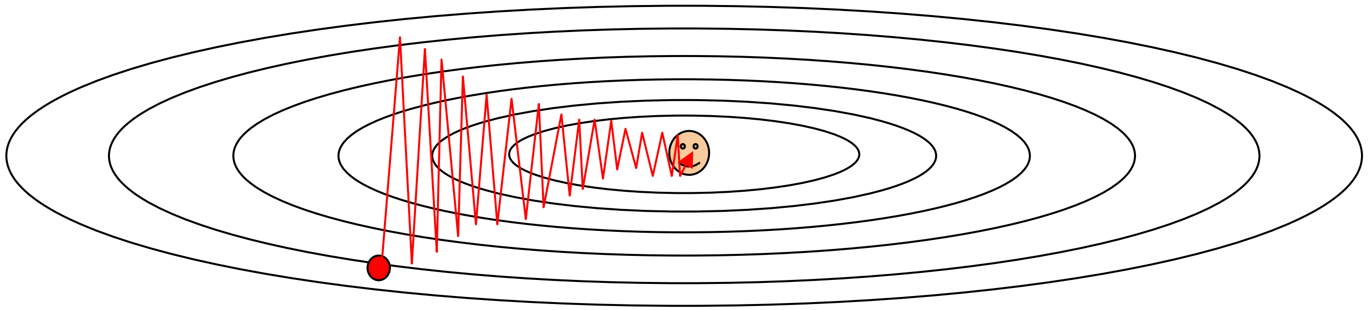
Local minima



Saddle points



Poor conditioning



Source: [J. Johnson]

SGD with momentum

- Goal: move faster in directions with consistent gradient, avoid oscillating in directions with large but inconsistent gradients

Standard SGD

SGD with momentum

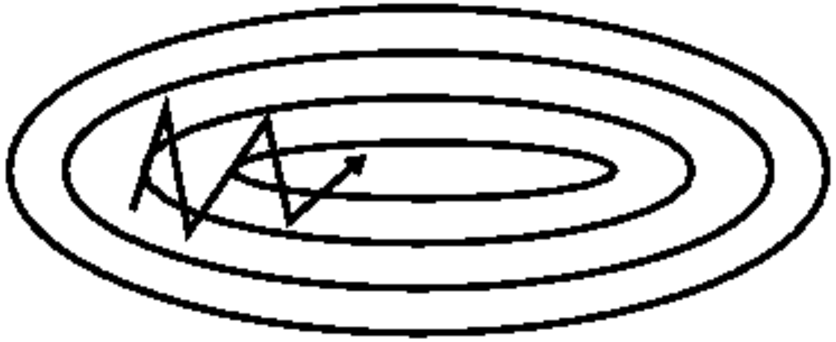


Image source

SGD with momentum

- Introduce a "momentum" variable m and associated "friction" coefficient β:

$$m \leftarrow \beta m - \eta \nabla L$$
$$w \leftarrow w + m$$

  - Typically start with $\beta = 0.5$, gradually increase over time

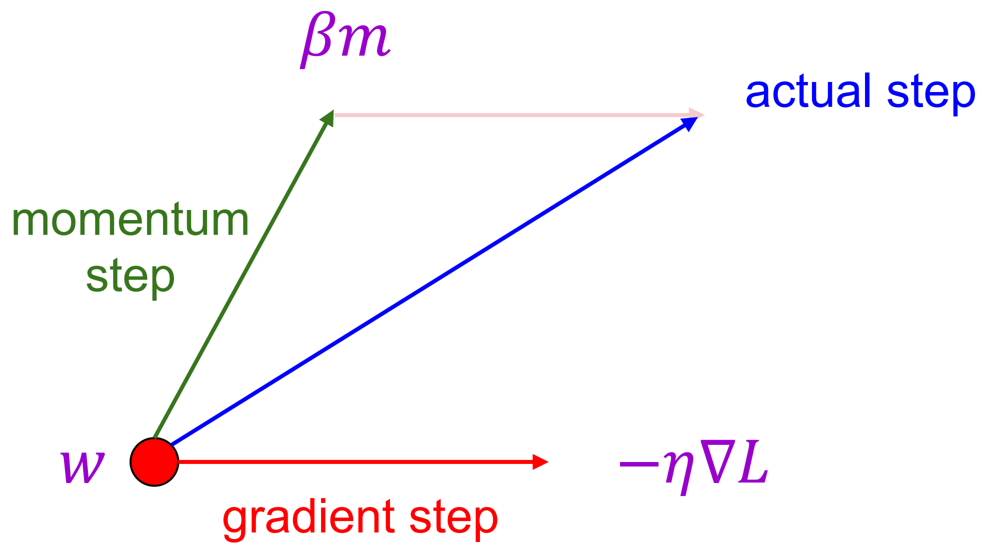


Image source

Adagrad: Adaptive per-parameter learning rates

- Keep track of history of gradient magnitudes, scale the learning rate for each parameter based on this history
- For each dimension k of the weight vector:

$$v^{(k)} \leftarrow v^{(k)} + \left(\frac{\partial L}{\partial w^{(k)}}\right)^2$$

Update running sum of squared magnitudes of gradient w.r.t. kth weight

$$w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{v^{(k)}} + \epsilon} \frac{\partial L}{\partial w^{(k)}}$$

Scale learning rate for kth weight by inverse of the magnitude, update kth weight

- Parameters with small gradients get large updates and vice versa
- Problem: long-ago gradient magnitudes are not “forgotten” so learning rate decays too quickly

J. Duchi, Adaptive subgradient methods for online learning and stochastic optimization, JMLR 2011

RMSProp

- Introduce decay factor β (typically ≥ 0.9) to downweight past history exponentially:

$$v^{(k)} \leftarrow \beta v^{(k)} + (1-\beta)\left(\frac{\partial L}{\partial w^{(k)}}\right)^2$$

$$w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{v^{(k)}} + \epsilon} \frac{\partial L}{\partial w^{(k)}}$$

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Adam: Combine RMSProp with momentum

- Update momentum:

$$m \leftarrow \beta_1 m + (1 - \beta_1)\nabla L$$

- For each dimension k of the weight vector:

$$v^{(k)} \leftarrow \beta_2 v^{(k)} + (1 - \beta_2)\left(\frac{\partial L}{\partial w^{(k)}}\right)^2$$

$$w^{(k)} \leftarrow w^{(k)} - \frac{\eta}{\sqrt{v^{(k)}} + \epsilon} m^{(k)}$$

- Full algorithm includes *bias correction* to account for m and v starting at 0: $\hat{m} = \frac{m}{1-\beta_1^t}$, $\hat{v} = \frac{v}{1-\beta_2^t}$ (t is the timestep)
- Default parameters from paper are reputed to work well for many models: $\beta_1 = 0.9,\ \beta_2 = 0.999,\ \eta = 1e - 3,\ \epsilon = 1e - 8$

D. Kingma and J. Ba, [Adam: A method for stochastic optimization](), ICLR 2015

Which optimizer to use in practice?

- Adaptive methods tend to reduce initial training error faster than SGD and are “safer”
  - Andrej Karpathy: *“In the early stages of setting baselines I like to use Adam with a learning rate of 3e-4. In my experience Adam is much more forgiving to hyperparameters, including a bad learning rate. For ConvNets a well-tuned SGD will almost always slightly outperform Adam, but the optimal learning rate region is much more narrow and problem-specific.”*
  - Use Adam early in training, switch to SGD for later epochs?

Which optimizer to use in practice?

- Adaptive methods tend to reduce initial training error faster than SGD and are "safer"
- Some literature has reported problems with adaptive methods, such as failing to converge or generalizing poorly (Wilson et al. 2017, Reddi et al. 2018)
- More recent comparative study (Schmidt et al., 2021):
  "We observe that evaluating multiple optimizers with default parameters works approximately as well as tuning the hyperparameters of a single, fixed optimizer."

Outline

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive methods: SGD with momentum, RMSProp, Adam
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops

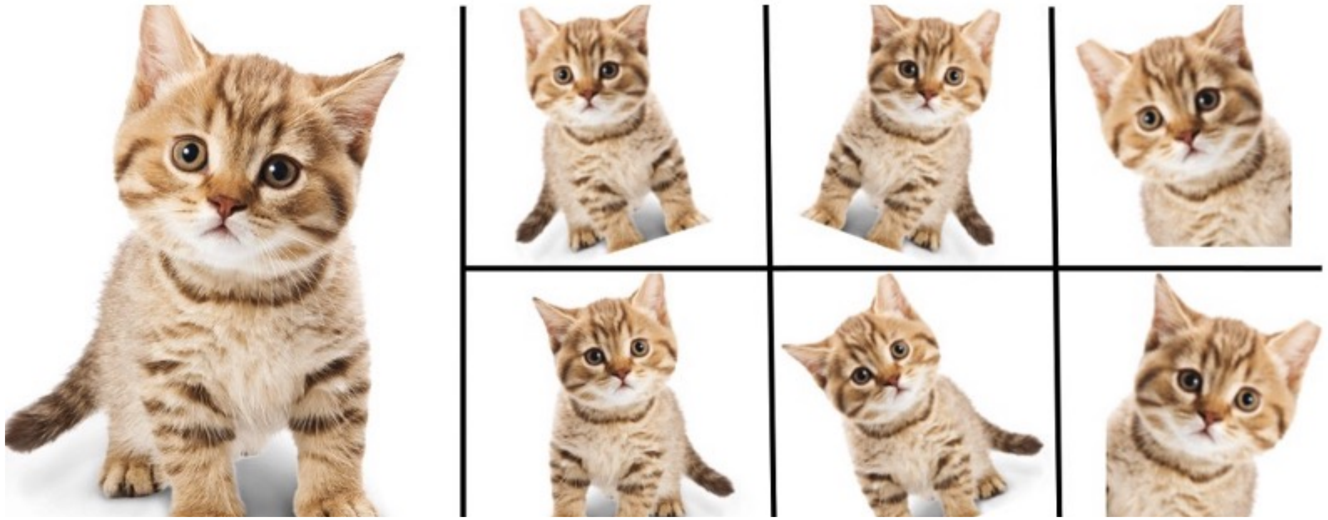


Image source

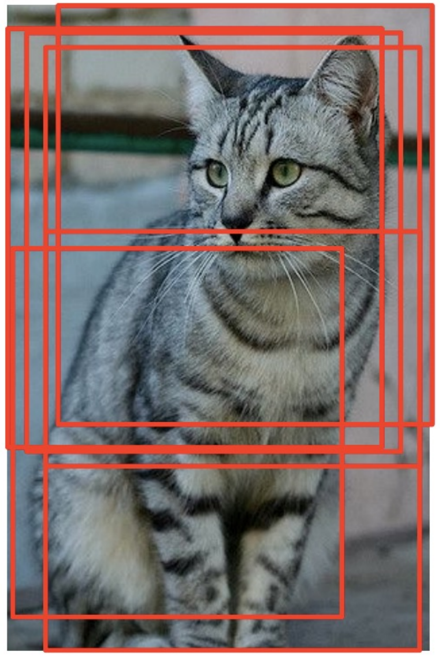


Image source

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
  - Photometric: color transformations

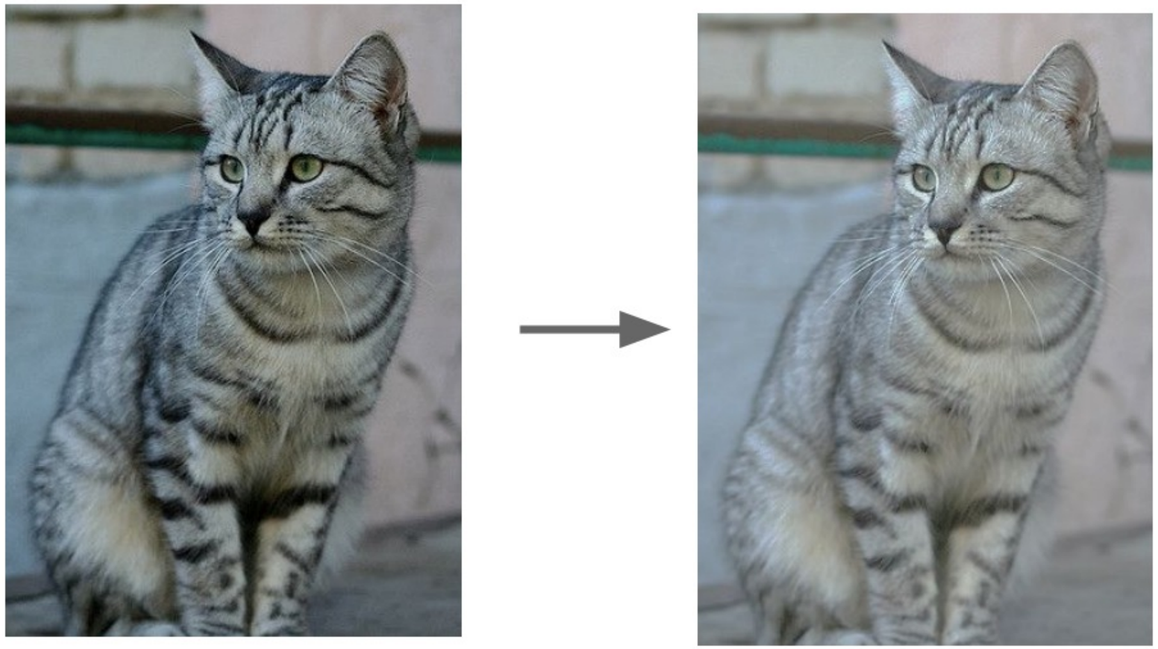


Image source

Data augmentation

- Introduce transformations not adequately sampled in the training data
  - Geometric: flipping, rotation, shearing, multiple crops
  - Photometric: color transformations
  - Other: add noise, compression artifacts, lens distortions, etc.

Image source

Data augmentation

- Introduce transformations not adequately sampled in the training data
- Limited only by your imagination and time/memory constraints!
- Avoid introducing artifacts
- Automatic augmentation strategies: AutoAugment, RandAugment

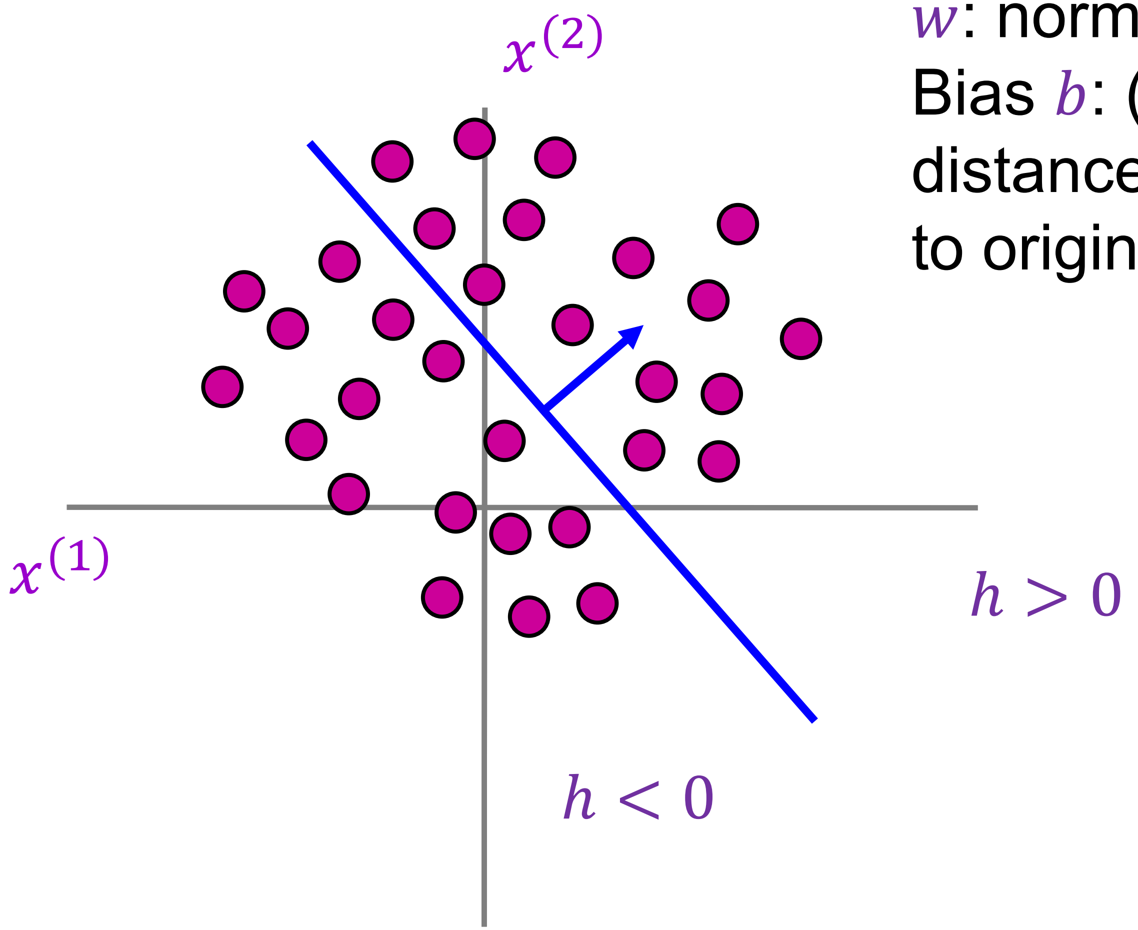
Data preprocessing

- Zero centering
  - Subtract *mean image* – all input images need to have the same resolution
  - Subtract *per-channel means* – images don’t need to have the same resolution
- Optional: rescaling – divide each value by (per-pixel or per-channel) standard deviation

- Be sure to apply the same transformation at training and test time!
  - Save training set statistics and apply to test data

The importance of preprocessing and initialization

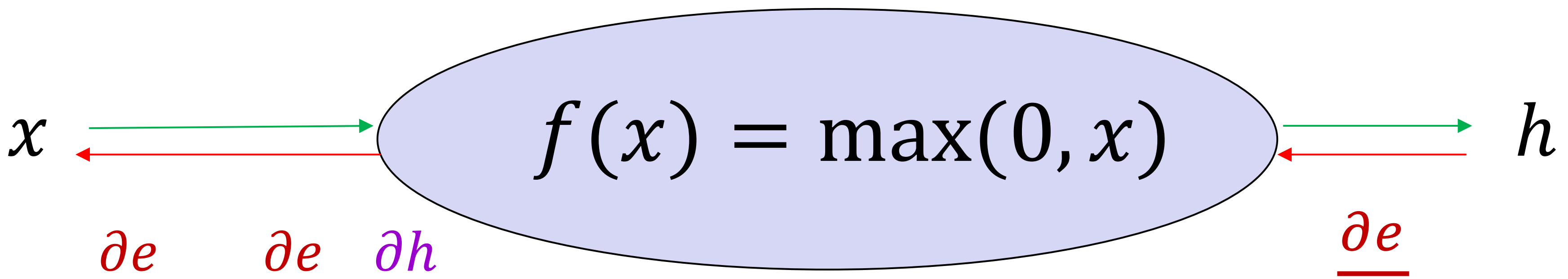
- Consider the behavior of a linear+ReLU unit: $h = \mathrm{ReLU}(w^T x + b)$



w: normal to a hyperplane
Bias b: (unnormalized) distance from hyperplane to origin

Review: Backward pass for ReLU

$$\frac{\partial h}{\partial x} = \mathbb{I}[x > 0]$$



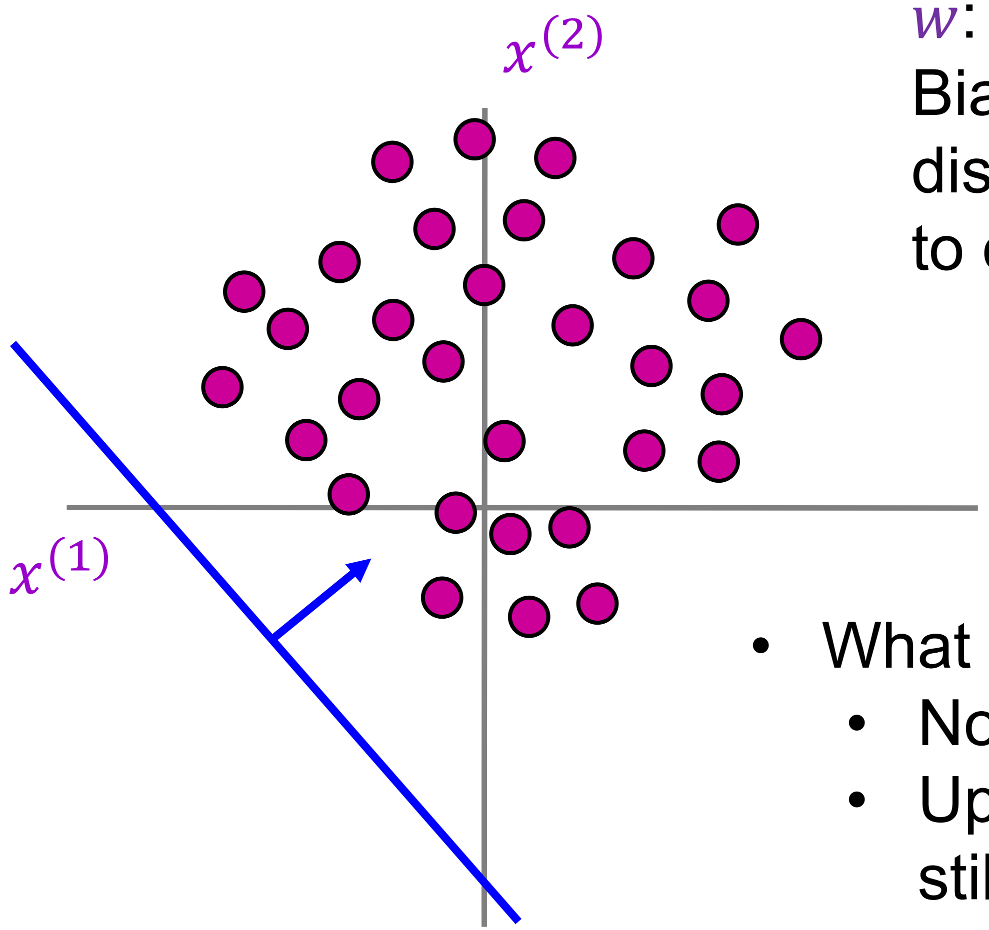
$$\frac{\partial e}{\partial h}$$

$$\frac{\partial e}{\partial x} = \frac{\partial e}{\partial h} \frac{\partial h}{\partial x}$$

$$\frac{\partial e}{\partial x} = \frac{\partial e}{\partial h} \mathbb{I}[x > 0]$$

The importance of preprocessing and initialization

Linear+ReLU unit: $h = \mathrm{ReLU}(w^T x + b)$

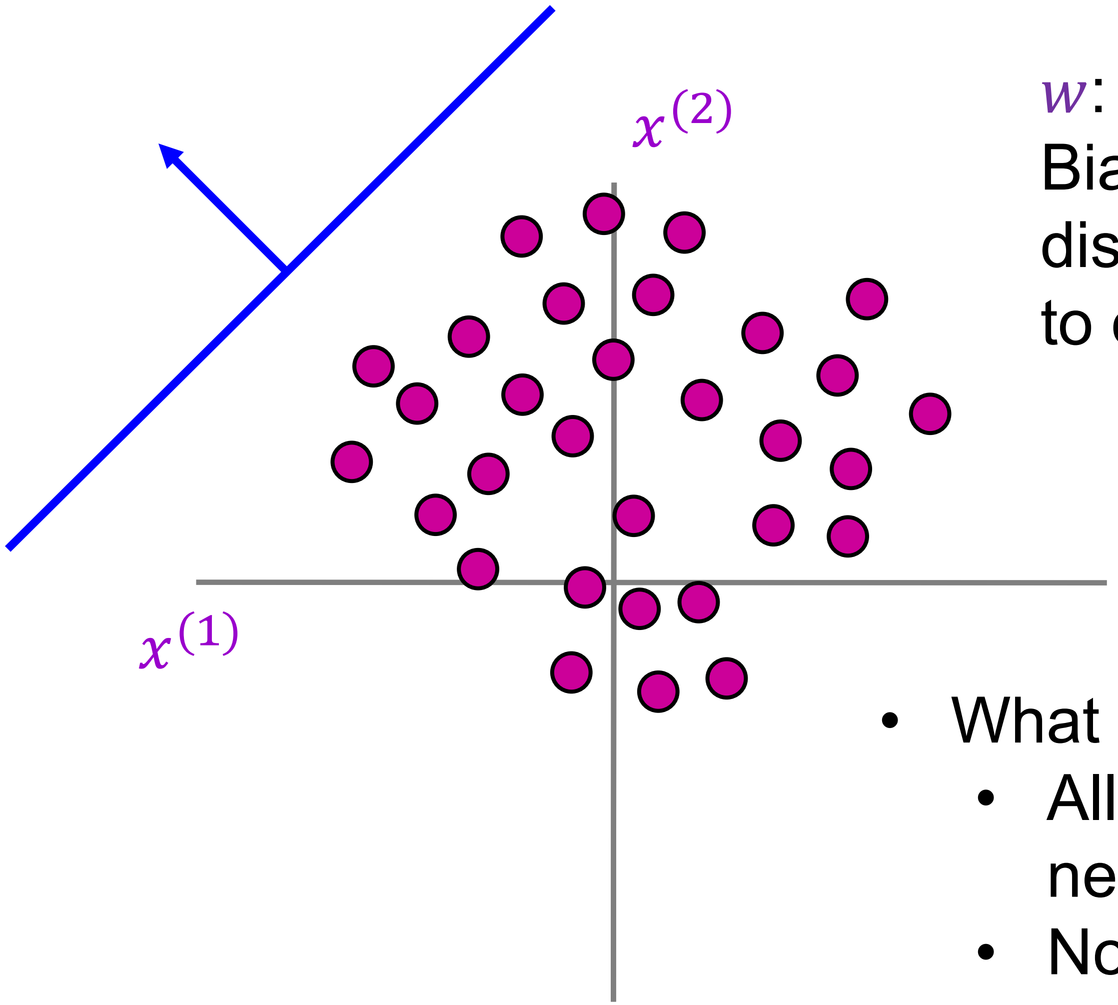


w: normal to a hyperplane
Bias b: (unnormalized) distance from hyperplane to origin

- What happens in this case?
  - Nonlinearity plays no role
  - Upstream gradients can still back-propagate

The importance of preprocessing and initialization

Linear+ReLU unit: $h = \mathrm{ReLU}(w^T x + b)$



w: normal to a hyperplane
Bias b: (unnormalized) distance from hyperplane to origin

- What happens in this case?
  - All inputs to ReLU are negative
  - No gradients propagate back – dead ReLU!

The importance of preprocessing and initialization

- What's wrong with initializing all weights to the same number (e.g., zero)?

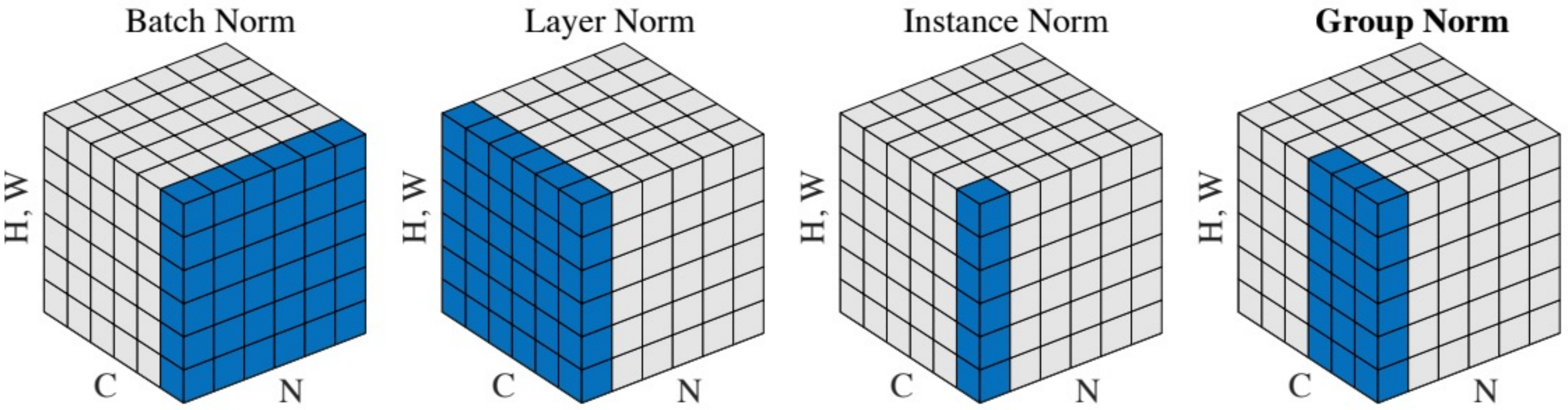
Weight initialization

- Typically: initialize to random values sampled from zero-mean Gaussian: $w \sim \mathcal{N}(0, \sigma^2)$
  - Standard deviation matters!
  - Key idea: avoid reducing or amplifying the variance of layer responses, which would lead to vanishing or exploding gradients
- Common heuristics:
  - Xavier initialization: $\sigma^2 = 1/n_{\text{in}}$ or $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$, where n_{in} and n_{out} are the numbers of inputs and outputs to a layer (Glorot and Bengio, 2010)
  - Kaiming initialization (goes with ReLU): $\sigma^2 = 2/n_{\text{in}}$ (He et al., 2015)
- Initializing biases: just set them to 0

More details: http://cs231n.github.io/neural-networks-2/#init

Normalization

- I omitted a crucial detail so far:
  - It is often useful to standardize statistics of hidden layers
  - through use of *normalization layers*
  - to mitigate vanishing / exploding gradients
  - when training deeper networks
- Many forms of normalization:



Batch normalization

- **Key idea:** shifting and rescaling are differentiable operations, so the network can *learn* how to best normalize the data
- Statistics of activations (outputs) from a given layer across the dataset can be approximated by statistics from a mini-batch

S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch normalization

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

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

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

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

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

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

S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch normalization

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

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

At test time (usually):

$$\cancel{\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i}$$ // ~~mini-batch~~ training set mean

$$\cancel{\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2}$$ // ~~mini-batch~~ training set variance

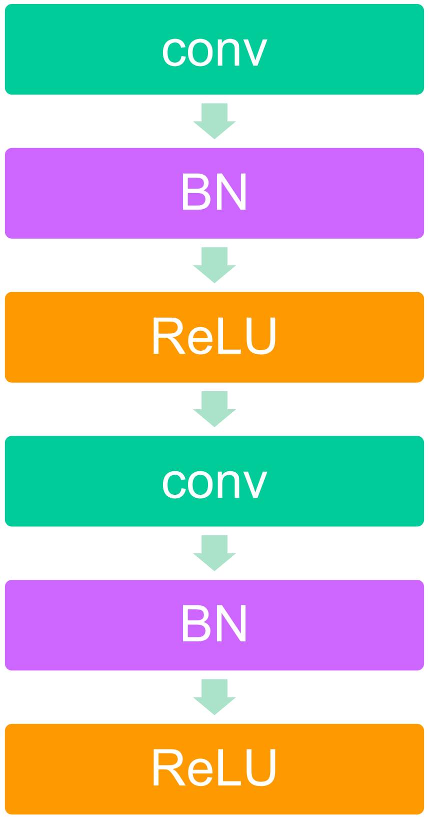
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$$ // normalize

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

S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch normalization

- Common configuration: insert BN layers right after conv or FC layers, before ReLU nonlinearity (but this is purely empirical)



S. Ioffe, C. Szegedy, [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](), ICML 2015

Batch normalization

- Benefits
  - Prevents exploding and vanishing gradients
  - Keeps most activations away from saturation regions of non-linearities
  - Accelerates convergence of training
  - Makes training more robust w.r.t. hyperparameter choice, initialization
- Pitfalls
  - Behavior depends on composition of mini-batches, can lead to hard-to-catch bugs if there is a mismatch between training and test regime (example)
  - Doesn’t work well for small mini-batch sizes
  - Cannot be used for certain types of models (recurrent models, transformers)

Batch Normalization (Results)

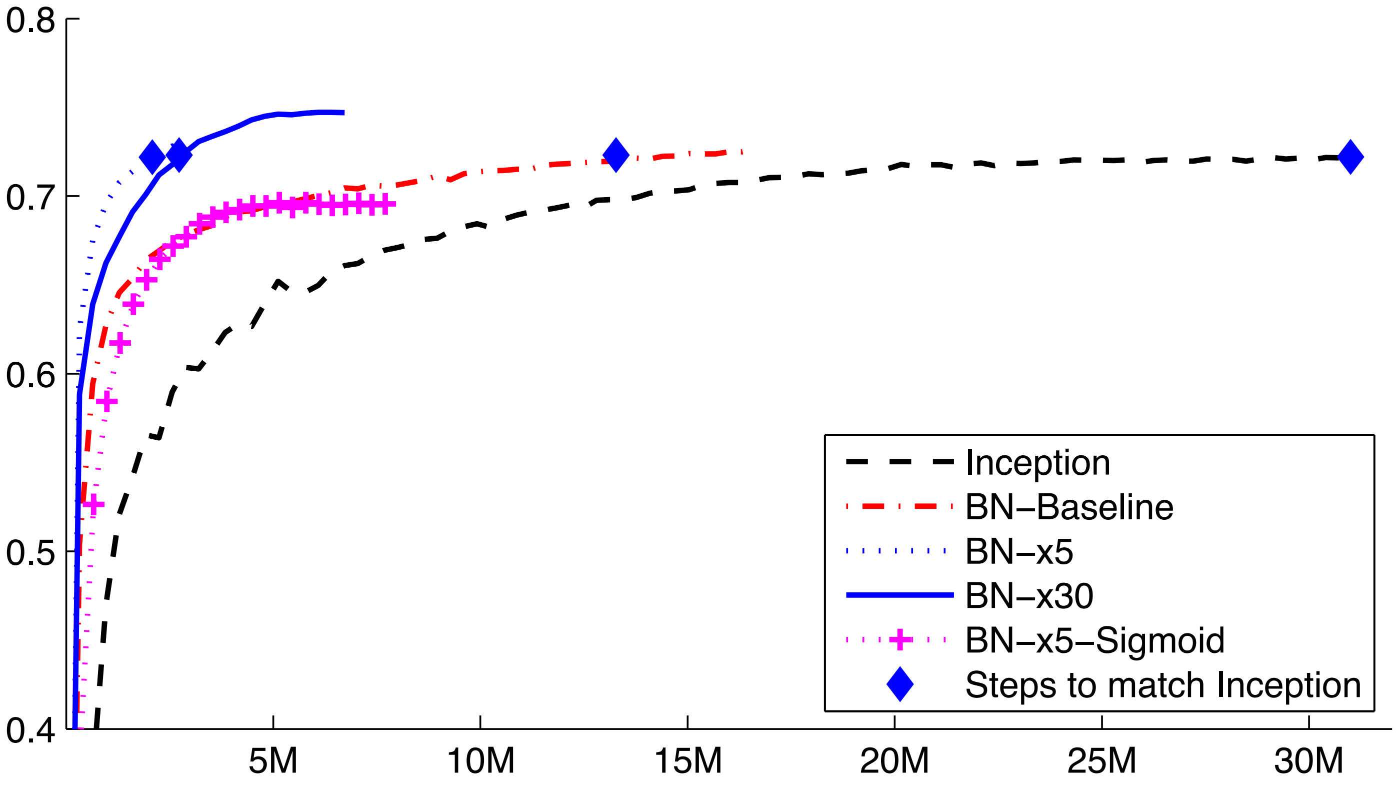
Inception: the network described at the beginning of Section 4.2, trained with the initial learning rate of 0.0015.

BN-Baseline: Same as Inception with Batch Normalization before each nonlinearity.

BN-x5: Inception with Batch Normalization and the modifications in Sec. 4.2.1. The initial learning rate was increased by a factor of 5, to 0.0075. The same learning rate increase with original Inception caused the model parameters to reach machine infinity.

BN-x30: Like *BN-x5*, but with the initial learning rate 0.045 (30 times that of Inception).

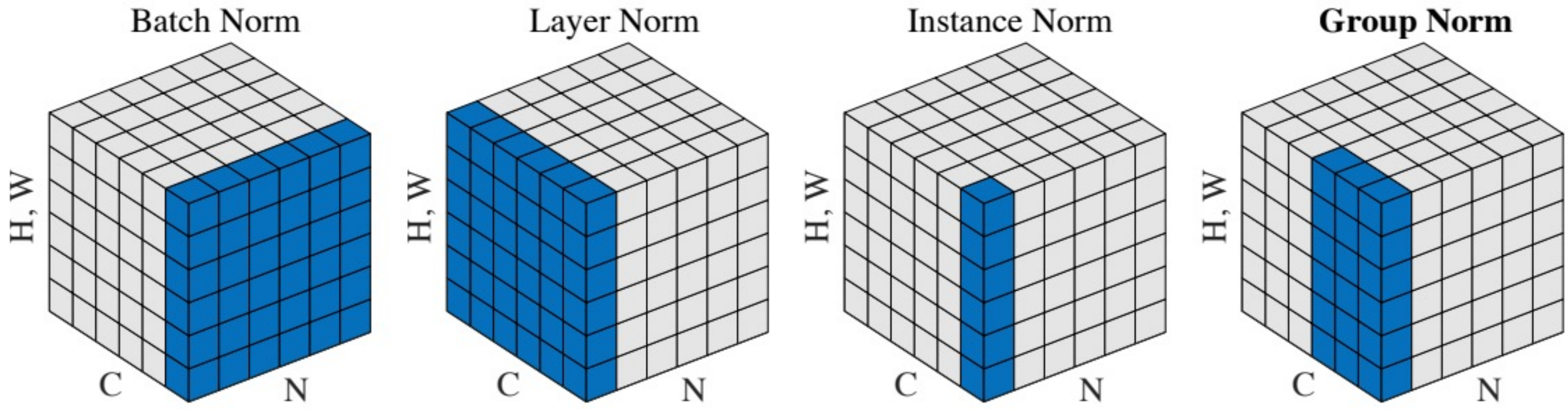
BN-x5-Sigmoid: Like *BN-x5*, but with sigmoid nonlinearity $g(t) = \frac{1}{1+\exp(-x)}$ instead of ReLU. We also attempted to train the original Inception with sigmoid, but the model remained at the accuracy equivalent to chance.



S. Iofffe and C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arXiv 2015

Other types of normalization

- Layer normalization (Ba et al., 2016)
- Instance normalization (Ulyanov et al., 2017)
- Group normalization (Wu and He, 2018)
- Weight normalization (Salimans et al., 2016)



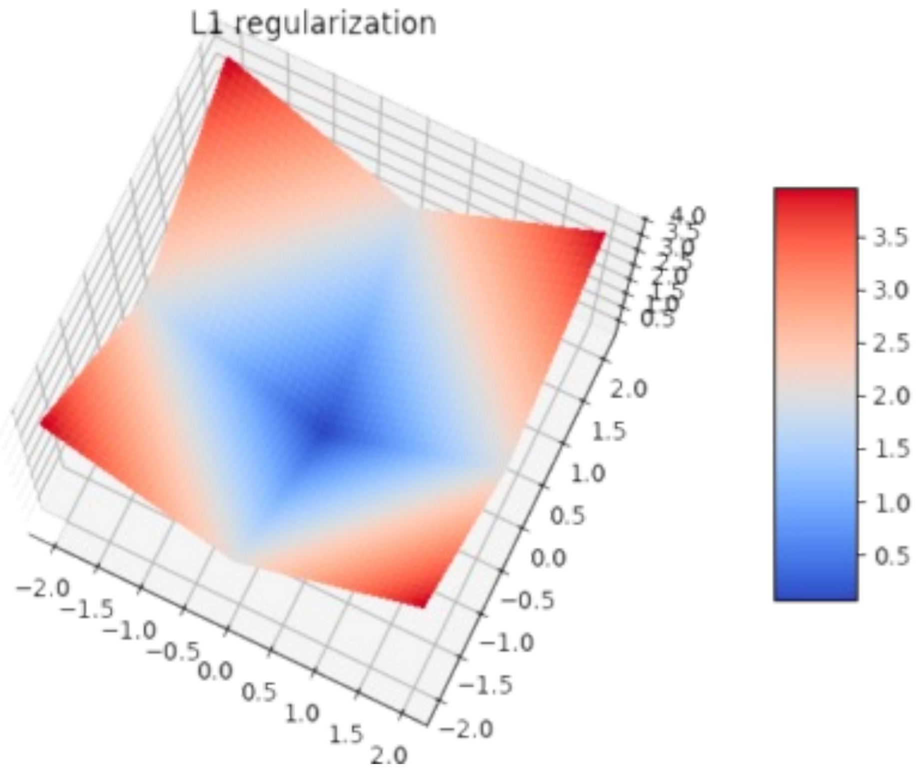
Y. Wu and K. He, Group Normalization, ECCV 2018

Outline

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive methods: SGD with momentum, RMSProp, Adam
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization
- Regularization

Regularization

- Techniques for controlling the capacity of a neural network to prevent overfitting – short of explicit reduction of the number of parameters
- Recall: classic regularization: L1, L2



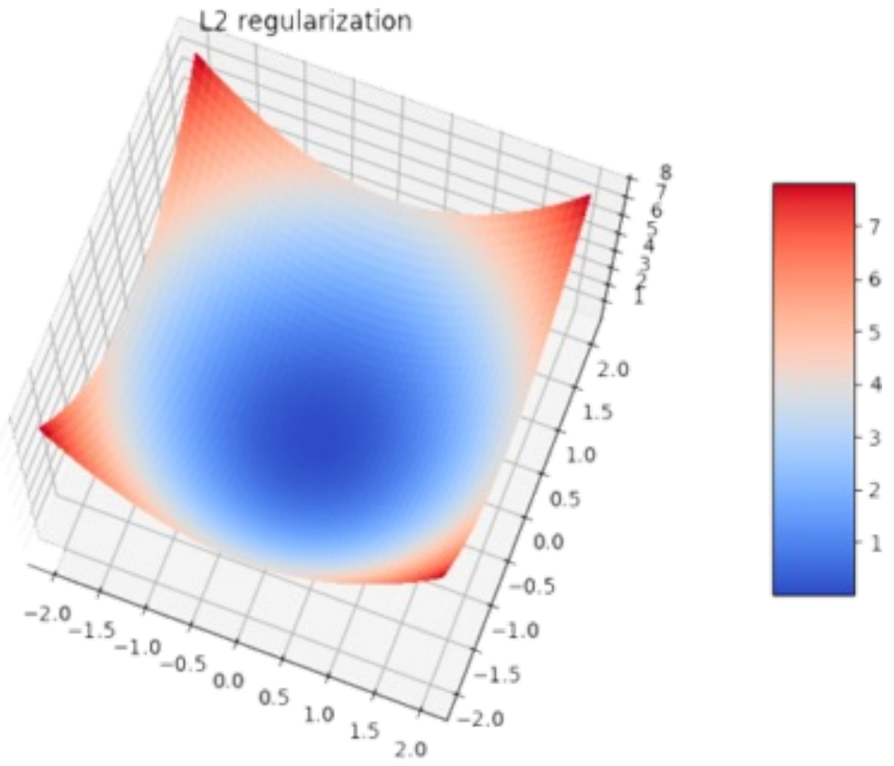


Image source

Weight decay

- Generic optimization step:

$$L(w) = L_{\text{data}}(w) + L_{\text{reg}}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = \text{optimizer}(g_t)$$
$$w_{t+1} = w_t - \eta s_t$$

- SGD with L2 regularization:

$$L(w) = L_{\text{data}}(w) + \frac{\lambda}{2}\|w\|^2$$
$$g_t = \nabla L_{\text{data}}(w_t) + \lambda w$$
$$\begin{aligned} w_{t+1} &= w_t - \eta g_t \\ &= (1 - \eta\lambda) w_t - \eta \nabla L_{\text{data}}(w_t) \end{aligned}$$

- Optimization with weight decay:

$$L(w) = L_{\text{data}}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = \text{optimizer}(g_t)$$
$$w_{t+1} = (1 - \eta\lambda) w_t - \eta s_t$$

I. Loshchilov and F. Hutter, Decoupled Weight Decay Regularization, ICLR 2019

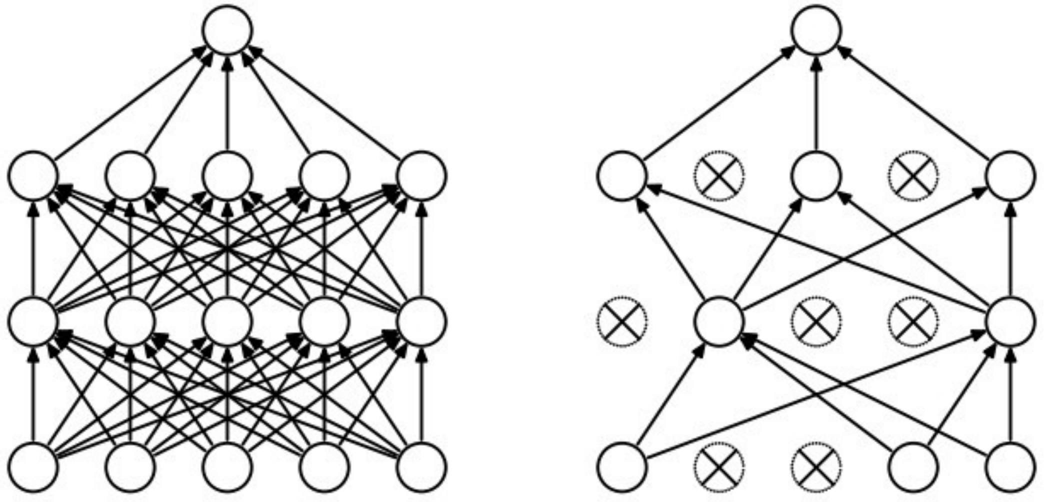
Adapted from J. Johnson

Other types of regularization

- Adding noise to the inputs
  - Recall motivation of max margin criterion
  - In simple scenario (linear model, quadratic loss, Gaussian noise), this is equivalent to weight decay
  - Data augmentation is a more general form of this
- Adding noise to the weights
- Label smoothing
  - When using softmax loss, replace hard 1 and 0 prediction targets with “soft” targets of $1 - \epsilon$ and $\frac{\epsilon}{C-1}$

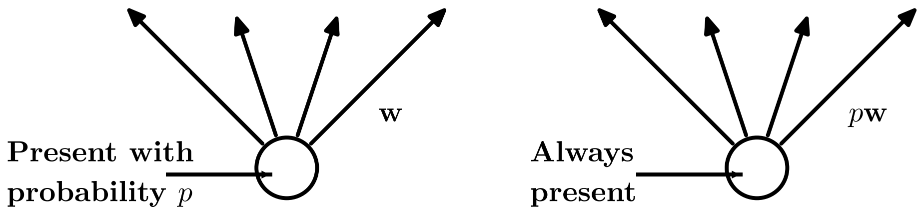
Dropout

- At training time, in each forward pass, turn off some neurons with probability p
- At test time, to have deterministic behavior, multiply output of neuron by p



(a) Standard Neural Net

(b) After applying dropout.



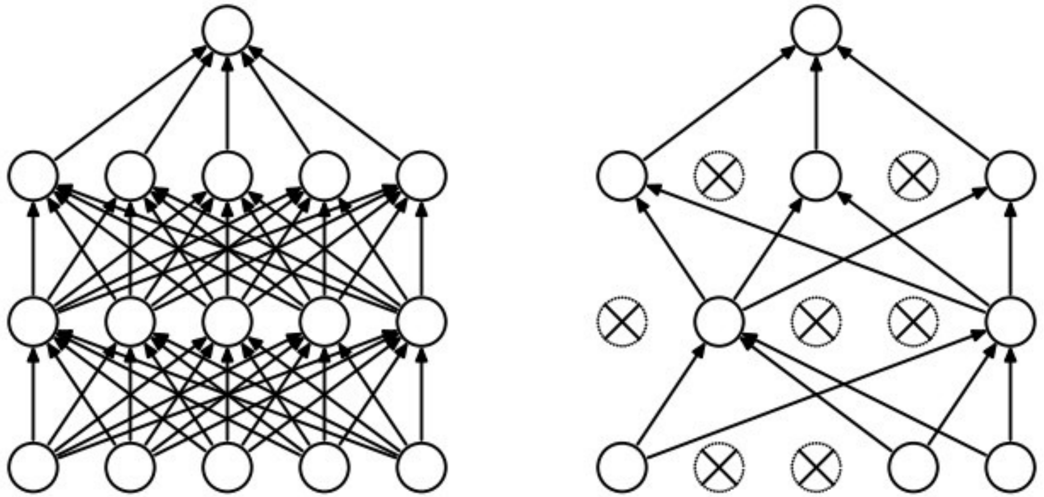
(a) At training time

(b) At test time

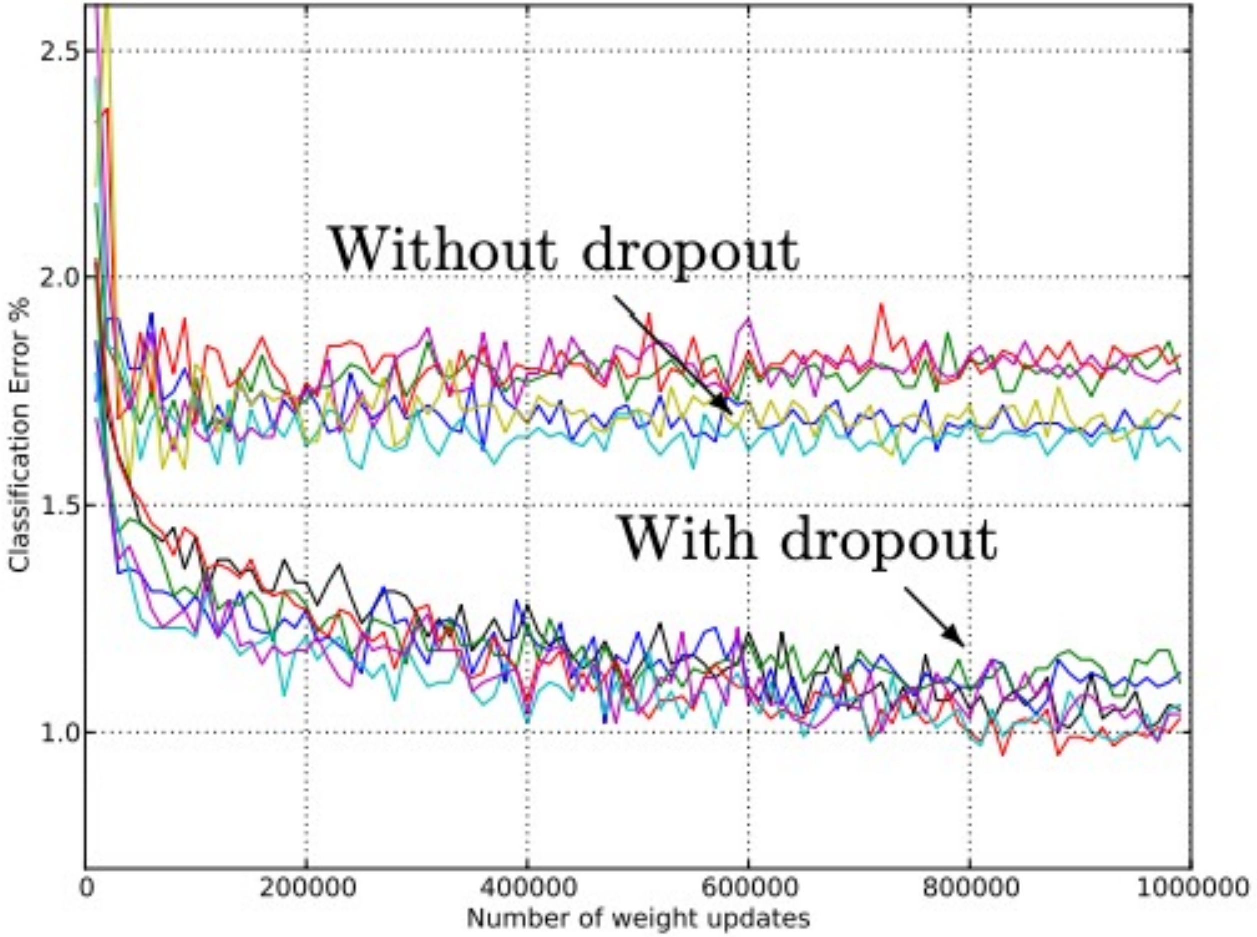
N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov.
Dropout: A Simple Way to Prevent Neural Networks from Overfitting. JMLR 2014

Dropout

- Intuitions
  - Prevent “co-adaptation” of units, increase robustness to noise
  - Train *implicit ensemble*



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



N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. [Dropout: A Simple Way to Prevent Neural Networks from Overfitting](). JMLR 2014

Current status of dropout

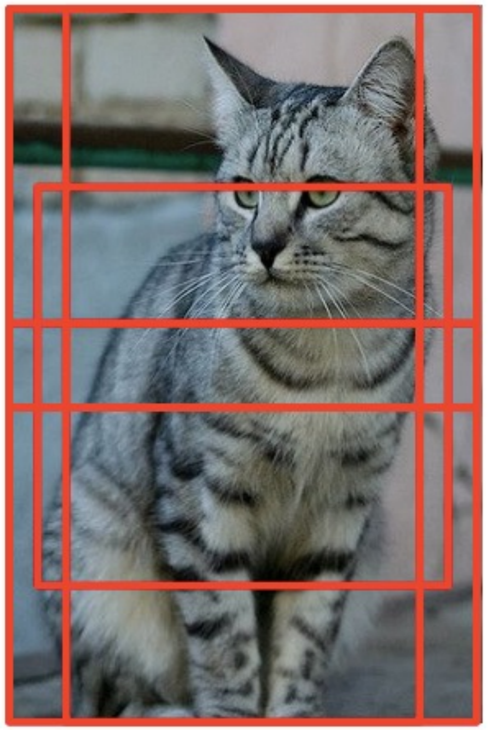
- Against
    - Slows down convergence
    - Made redundant by batch normalization or possibly even clashes with it
    - Unnecessary for larger datasets or with sufficient data augmentation
- In favor
    - Can still help for certain models and in certain situations: e.g., used in Wide Residual Networks

Outline

- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive methods: SGD with momentum, RMSProp, Adam
- Massaging the numbers
  - Data augmentation
  - Data preprocessing
  - Weight initialization
  - Batch normalization
- Regularization
- Test time: averaging predictions, ensembles

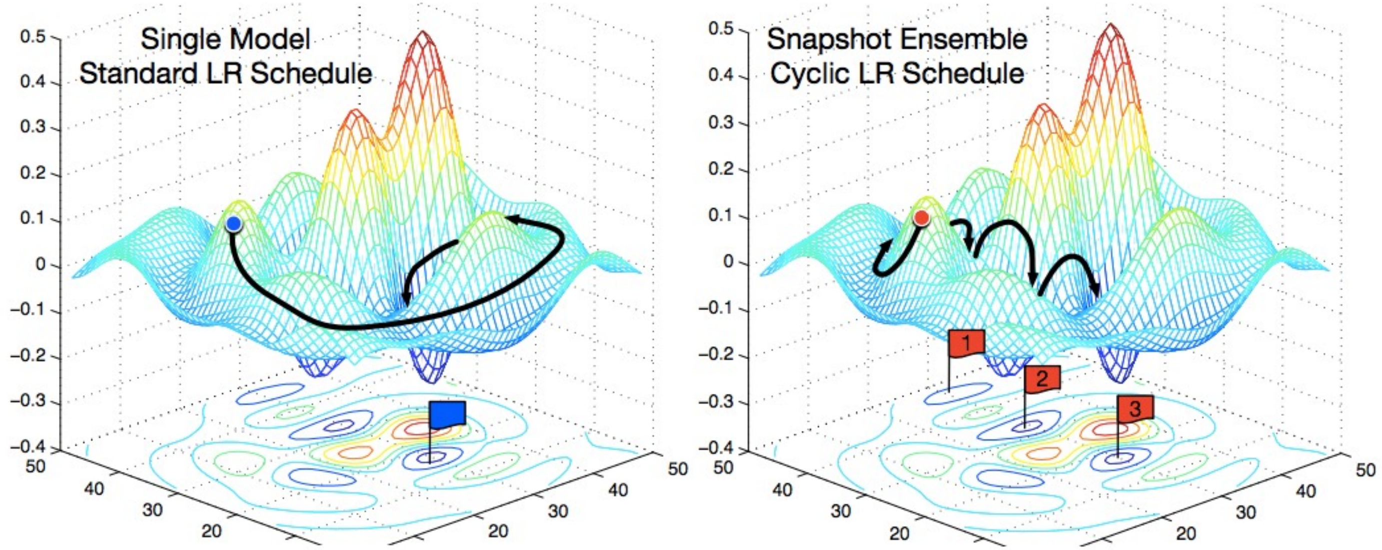
Test time

- Average predictions across multiple crops of test image
    - There is a more elegant way to do this with *fully convolutional networks* (FCNs)



Test time

- **Ensembles:** train multiple independent models, then average their predicted label distributions
  - Gives 1-2% improvement in most cases
  - Can take multiple snapshots of models obtained during training, especially if you *cycle* the learning rate (increase to jump out of local minima)



G. Huang et al., [Snapshot ensembles: Train 1, get M for free](), ICLR 2017

Important Considerations

1. Data
2. Supervision
3. Loss functions
4. Optimization / Initialization
5. Inductive Bias

Development Process

1. Collect lots of labeled data
2. Setup network architecture
3. Setup loss function
4. Sanity checks
    1. Is your data correct?
    2. Can you overfit to a small set?
5. Hyperparameters
    1. Learning hyperparameters: batch size, learning rates, how much to train, regularization, optimizer.
    2. Architectural hyper-parameters: Non-linearities, #layers, #neurons, loss functions.
6. Hacking
    1. Reducing iteration time
    2. Maximizing GPU utilization

Some take-aways

- Training neural networks is still a black art
- Process requires close “babysitting”
- For many techniques, the reasons why, when, and whether they work are in active dispute – read everything but don’t trust anything
- It all comes down to (principled) trial and error
- Further reading: A. Karpathy, A recipe for training neural networks