

Neural network training: Beyond the basics



Outline

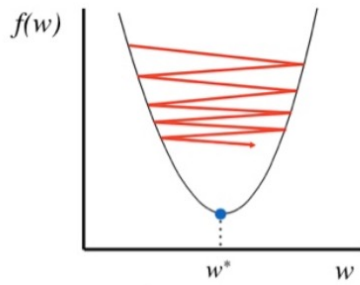
- Optimization
 - Mini-batch SGD
 - Learning rate decay
 - Diagnosing learning curves
 - Adaptive optimization methods
- Massaging the numbers
 - Data augmentation
 - Data preprocessing
 - Weight initialization
 - Batch normalization
- Regularization
- Test time: averaging predictions, ensembles
- Student-teacher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

Review: Mini-batch SGD

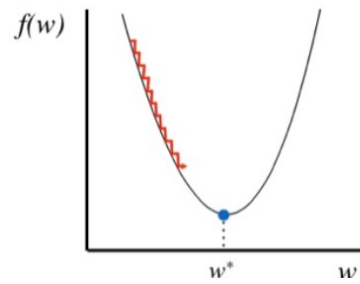
- Iterate over epochs:
 - Group data into mini-batches of size b
 - Compute gradient of the loss for the mini-batch $(x_1, y_1), \dots, (x_b, y_b)$:
$$\nabla \hat{L} = \frac{1}{b} \sum_{i=1}^b \nabla l(w, x_i, y_i)$$
 - Update parameters:
$$w \leftarrow w - \eta \nabla \hat{L}$$
 - Check for convergence, decide whether to decay learning rate
- Hyperparameters: mini-batch size, learning rate decay schedule, deciding when to stop

Setting the learning rate

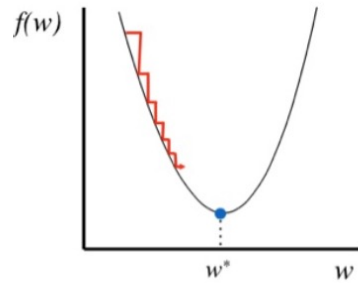
Too high



Too low



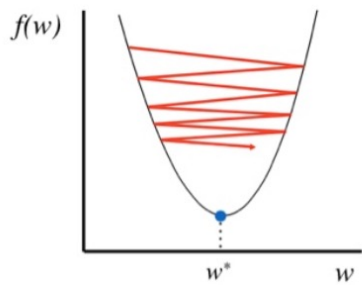
Want: good *decay schedule*



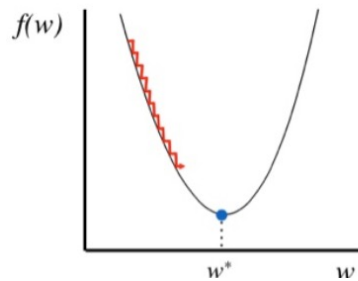
[Figure source](#)

Setting the learning rate

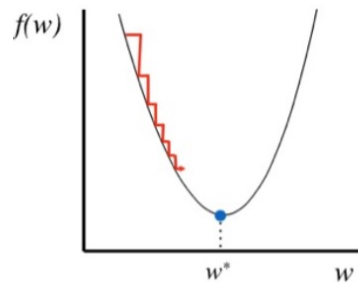
Too high



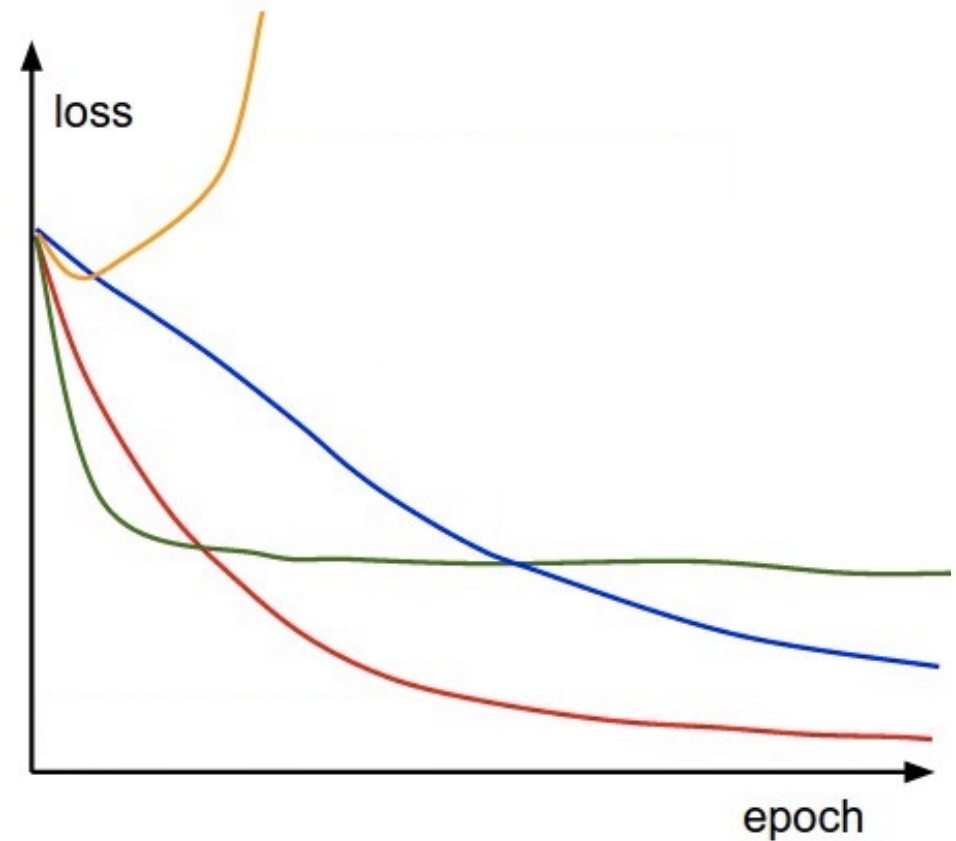
Too low



Want: good *decay schedule*

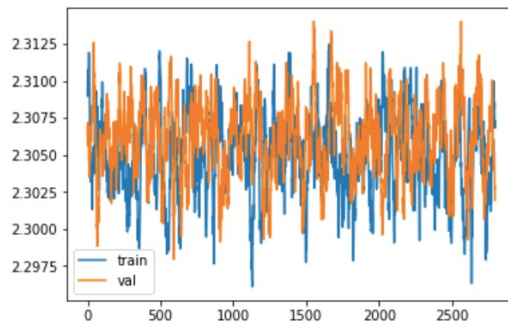


[Figure source](#)

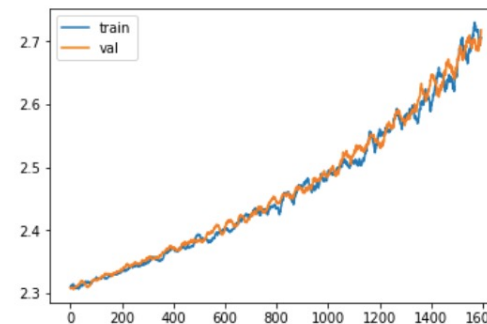


Source: [Stanford CS231n](#)

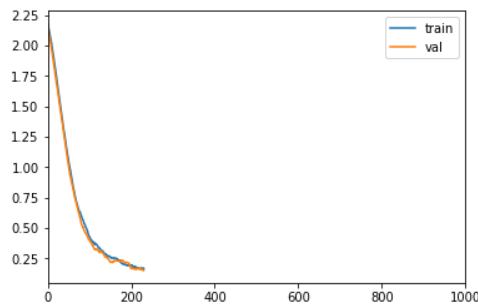
Diagnosing learning curves: Obvious problems



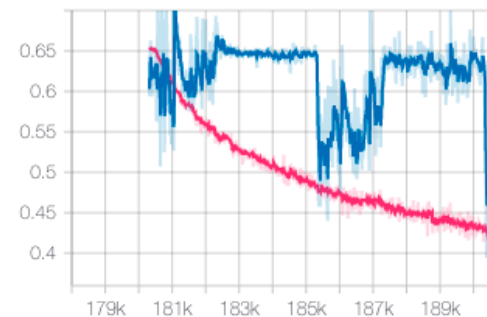
Not training
Bug in update calculation?



Error increasing
Bug in update calculation?



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



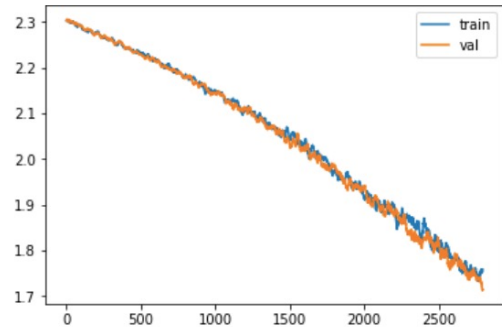
Weird cyclical patterns in loss:
Data not shuffled

Shuffling off

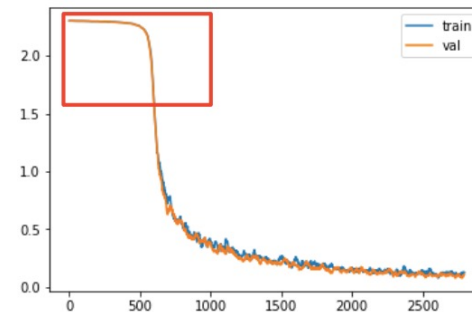
Shuffling on

Source: [Stanford CS231n](https://stanford.edu/cs231n/)

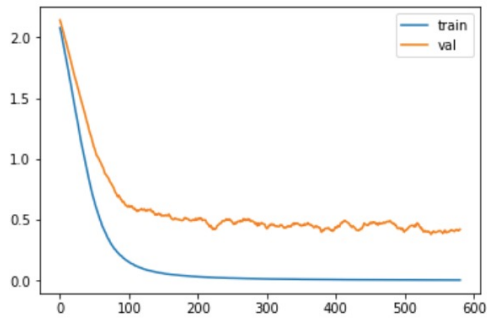
Diagnosing learning curves: Subtler behaviors



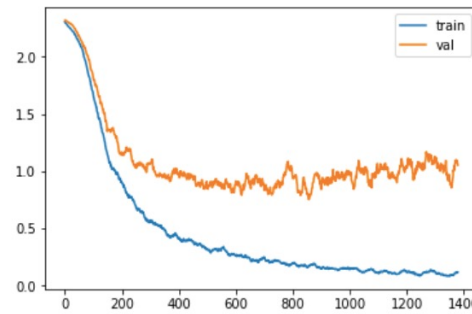
Not converged yet
Keep training, possibly increase learning rate



Slow start
Bad initialization?



Possible overfitting



Definite overfitting

Source: [Stanford CS231n](https://stanford.edu/)

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 or inverse sqrt: $\eta_t = \eta_0 / (1 + kt)$ or $\eta_t = \eta_0 / \sqrt{t}$
 - Linear: $\eta_t = \eta_0 (1 - t/T)$, where T is the total number of epochs
 - Cosine: $\eta_t = \frac{1}{2} \eta_0 (1 + \cos(t\pi/T))$
- 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
 - **Warmup:** sometimes it is beneficial to start with a low learning rate and slowly increase it before starting to decay ([Goyal et al., 2018](#))

A typical phenomenon

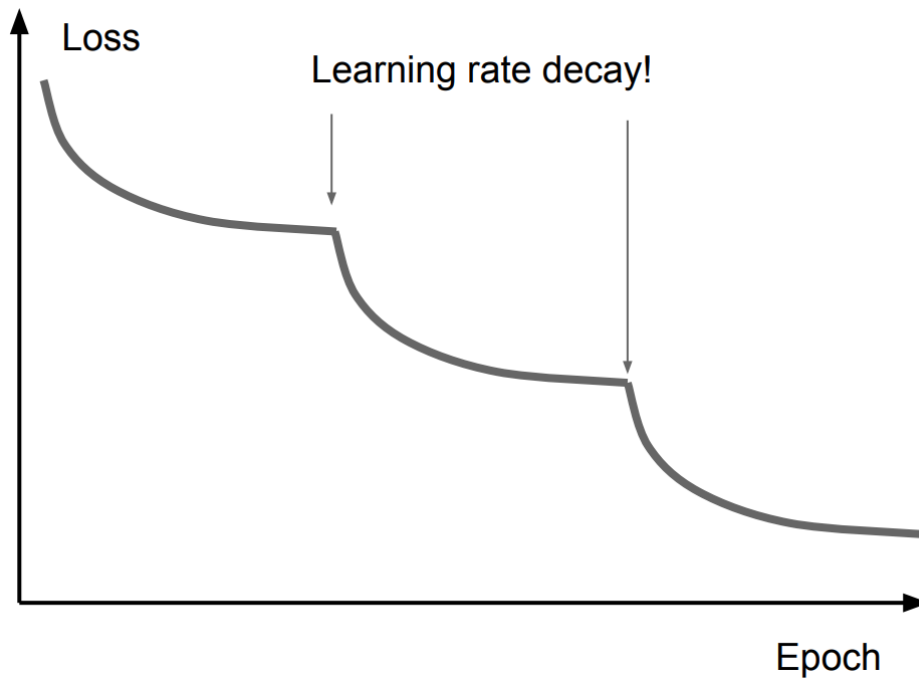


Image source: [Stanford CS231n](#)

Possible explanation



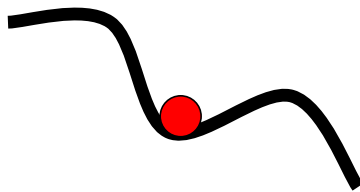
[Image source](#)

Outline

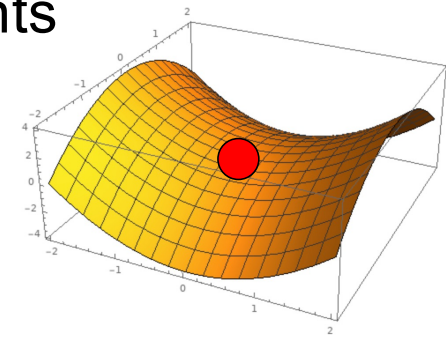
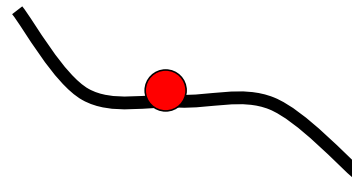
- Optimization
 - Mini-batch SGD
 - Learning rate decay
 - Diagnosing learning curves
 - Adaptive optimization methods

Where does SGD run into trouble?

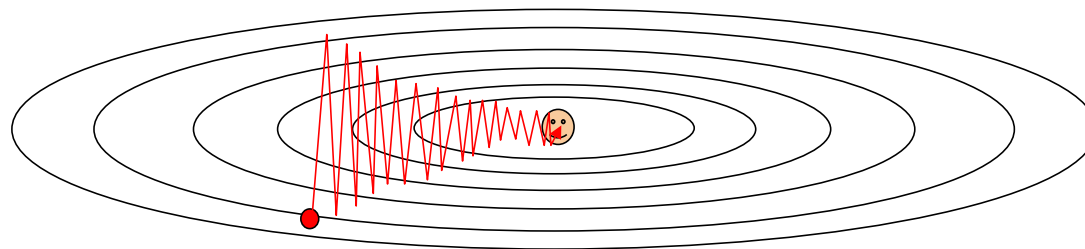
Local minima



Saddle points



Poor conditioning



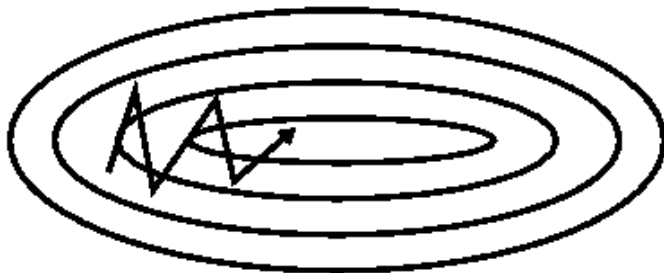
SGD with momentum

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

Standard SGD



SGD with momentum



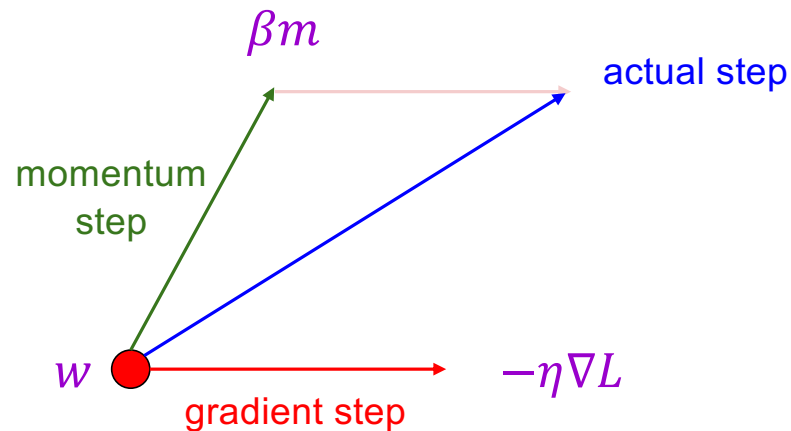
[Image source](#)

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](#)

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:

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

Update running sum of squared magnitudes of gradient w.r.t. k th weight (typically $\beta \geq 0.9$)

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

- Parameters with small gradients get large updates and vice versa

J. Duchi, [Adaptive subgradient methods for online learning and stochastic optimization](#), JMLR 2011

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

Adam: Combine momentum with per-parameter scaling

- 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?

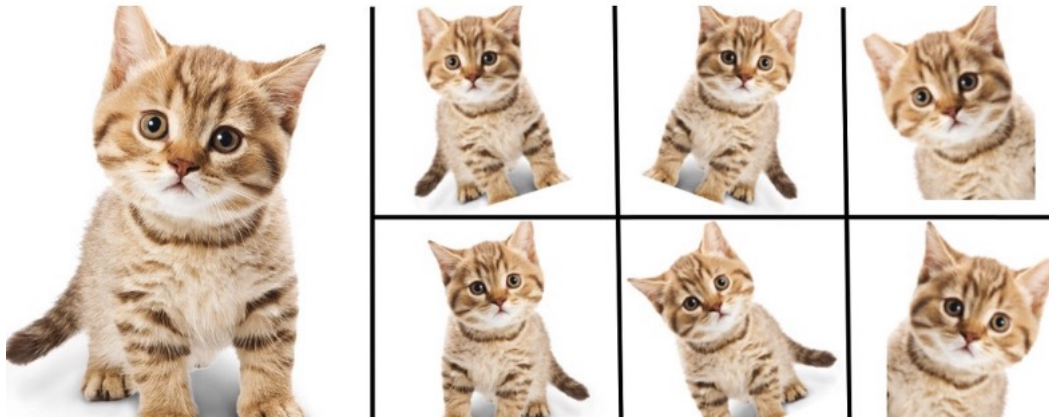
- Adaptive methods tend to reduce initial training error faster than SGD and are “safer”
- Some literature has reported problems with adaptive methods, such as failing to converge or generalizing poorly ([Wilson et al. 2017](#), [Reddi et al. 2018](#))
- More recent comparative study ([Schmidt et al.,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

Outline

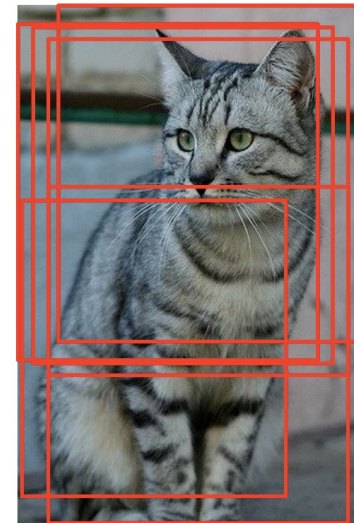
- Optimization
 - Mini-batch SGD
 - Learning rate decay
 - Diagnosing learning curves
 - Adaptive methods
- Massaging the numbers
 - Data augmentation
 - Data preprocessing
 - Weight initialization
 - Batch normalization

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.
 - 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

Weight 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>

Batch normalization

- The authors' intuition



[Image source](#), via Prajit Ramachandran

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

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

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

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

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

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

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

At test time (usually):

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

training set

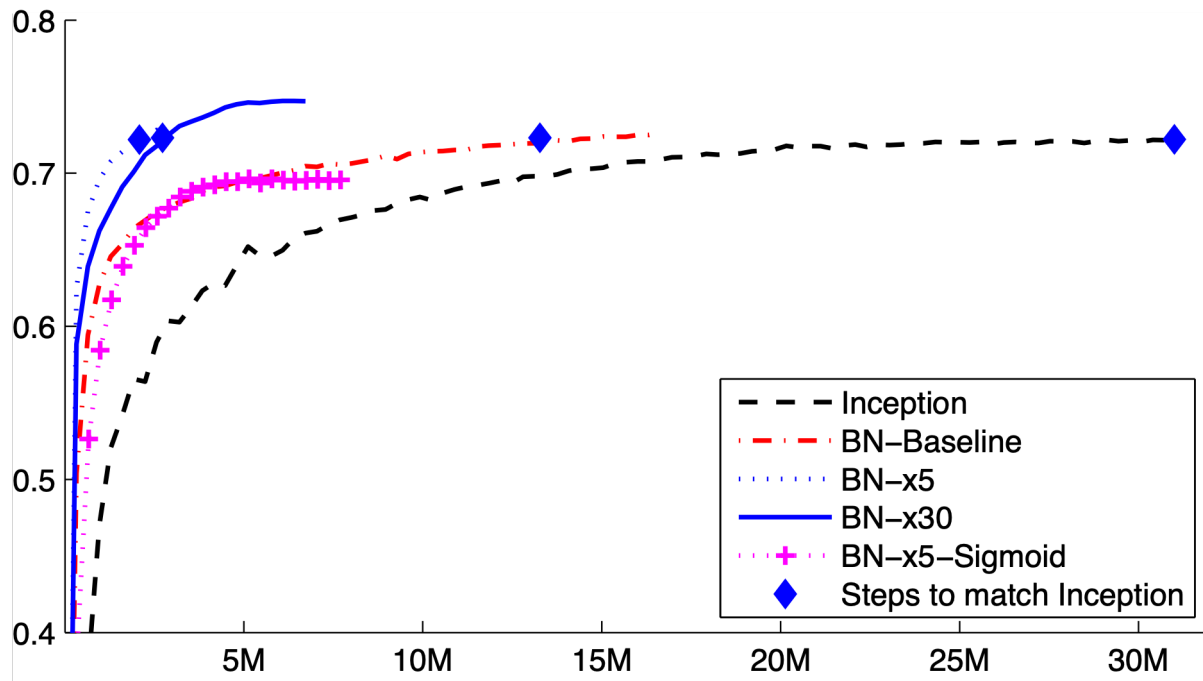
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ ~~mini batch~~ variance}$$

training set

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

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

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.

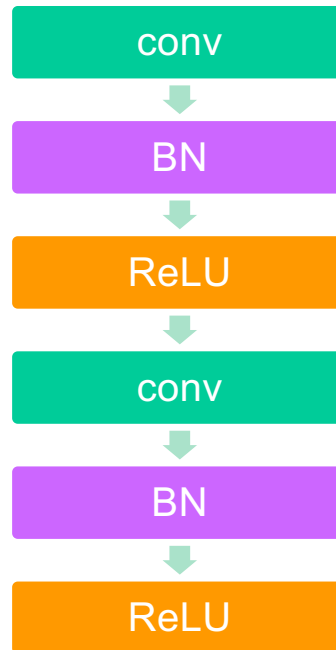
S. Ioffe, C. Szegedy, [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#), ICML 2015

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 settings ([example](#))
 - Doesn't work well for small mini-batch sizes
 - Cannot be used for certain types of models (recurrent models, transformers)

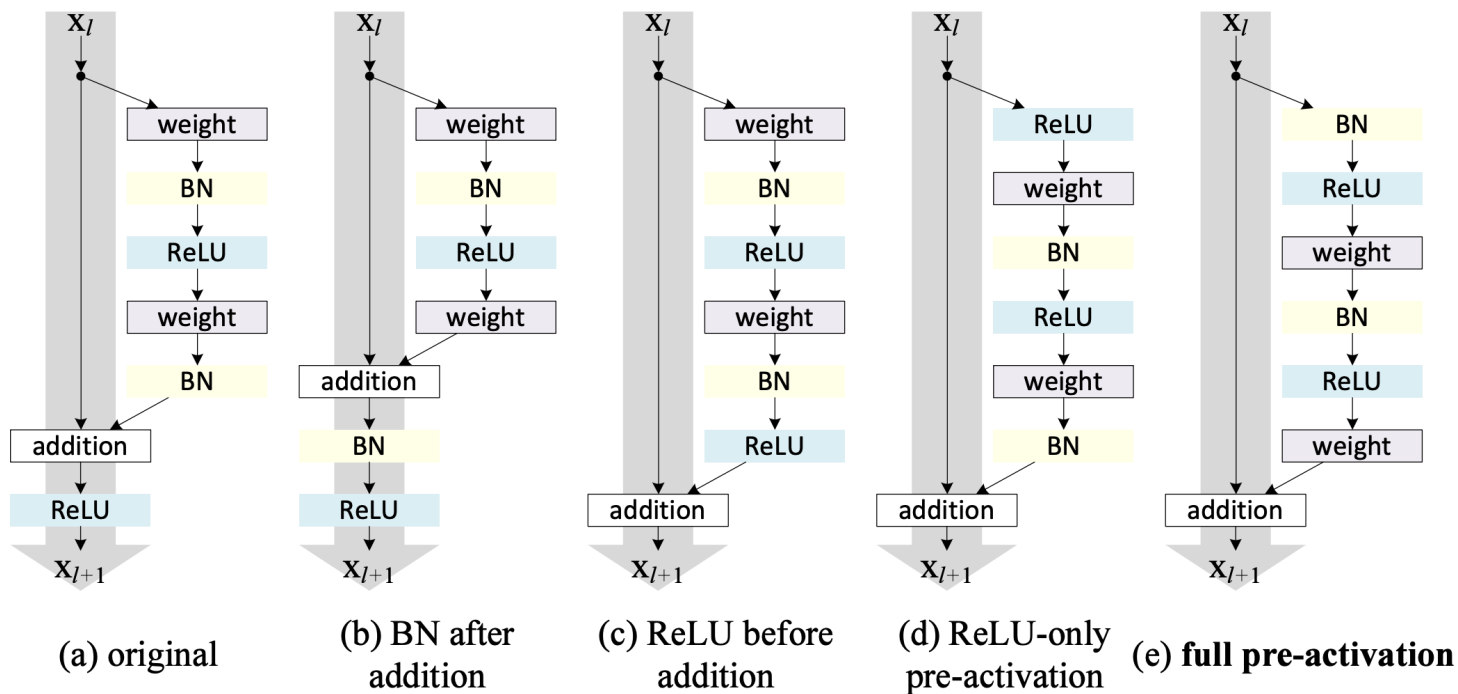
Where should BN be used in a network?

- BN paper recommends putting it before ReLU, but some practitioners [recommend the opposite](#)



Where should BN be used in a network?

- Detailed study of residual block design:

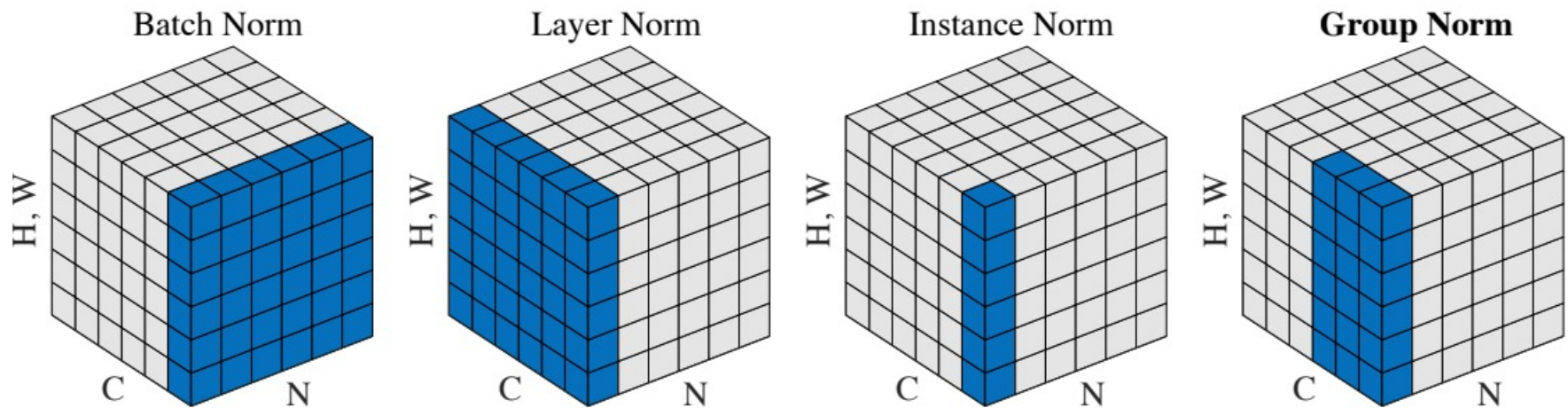


	ResNet-110	ResNet-164
(a)	6.61	5.93
(b)	8.17	6.50
(c)	7.84	6.14
(d)	6.71	5.91
(e)	6.37	5.46

K. He et al. [Identity Mappings in Deep Residual Networks](#). ECCV 2016

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)



[Figure source](#)

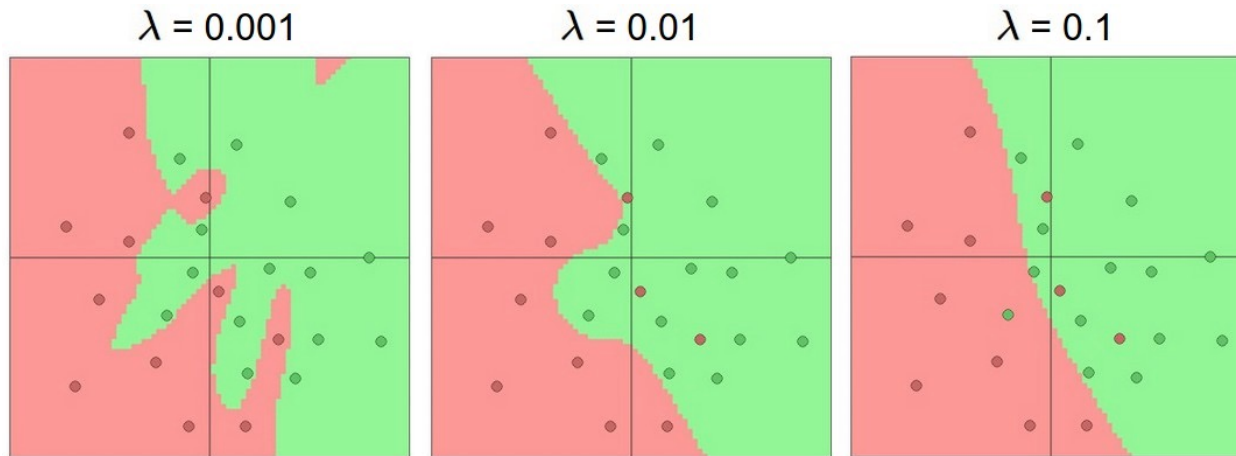
Outline

- Optimization
 - Mini-batch SGD
 - Learning rate decay
 - Diagnosing learning curves
 - Adaptive optimization methods
- Massaging the numbers
 - Data augmentation
 - Data preprocessing
 - Weight initialization
 - Batch normalization
- Regularization

What is regularization?

- Traditional viewpoint: regularization is a technique for controlling the capacity of a neural network to prevent overfitting (short of reducing the number of parameters)

Decision boundary with different values of regularization constant



What is regularization?

- ~~Traditional viewpoint: regularization is a technique for controlling the capacity of a neural network to prevent overfitting (short of reducing the number of parameters)~~
- Deep learning viewpoint: anything that helps you get 1-2% better performance on held-out data!

L2 regularization and weight decay

- SGD with L2 regularization:

$$L(w) = L_{\text{data}}(w) + \frac{\lambda}{2} \|w\|^2$$

$$g_t = \nabla L_{\text{data}}(w_t) + \lambda w$$

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

- Generic optimization step:

$$L(w) = L_{\text{data}}(w) + L_{\text{reg}}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

$$w_{t+1} = w_t - \eta s_t$$

- Optimization with decoupled weight decay (e.g., AdamW):

$$L(w) = L_{\text{data}}(w)$$

$$g_t = \nabla L(w_t)$$

$$s_t = \text{optimizer}(g_t)$$

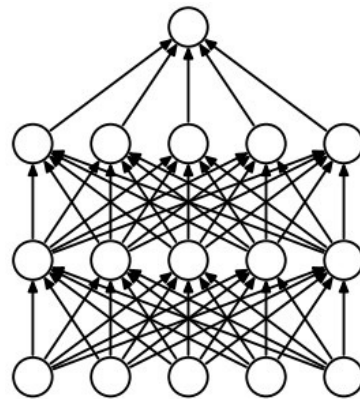
$$w_{t+1} = (1 - \eta \lambda) w_t - \eta s_t$$

Regularization by adding noise

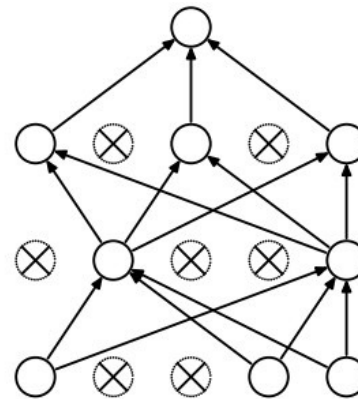
- Adding noise to the inputs
 - Recall motivation of max margin criterion
 - In simple scenario (linear model, quadratic loss, Gaussian noise), this is equivalent to weight decay
 - Data augmentation is a more general form of this
- Adding noise to the labels (within reason)
 - Recall label smoothing: when using softmax loss, replace hard 1 and 0 prediction targets with “soft” targets of $1 - \epsilon$ and $\frac{\epsilon}{C-1}$
- Adding noise to the weights
 - Lean into the “S” in SGD
 - [Stochastic gradient Langevin dynamics](#) (SLGD): add Gaussian noise deliberately based on stochastic sampling interpretation

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



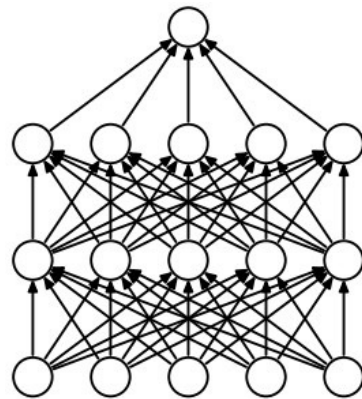
(b) After applying dropout.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov.

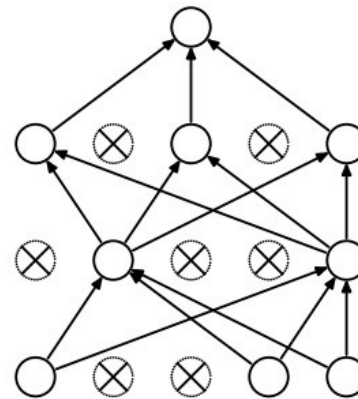
[Dropout: A Simple Way to Prevent Neural Networks from Overfitting.](#) JMLR 2014

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

Should you use dropout?

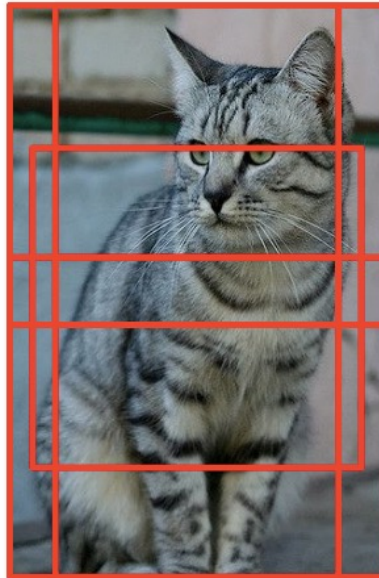
- Against
 - Slows down convergence
 - Made redundant by batch normalization or possibly even [clashes with it](#)
 - Unnecessary for larger datasets or with sufficient data augmentation
- In favor
 - Can still help for certain models and in certain situations: e.g., used in Wide Residual Networks

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

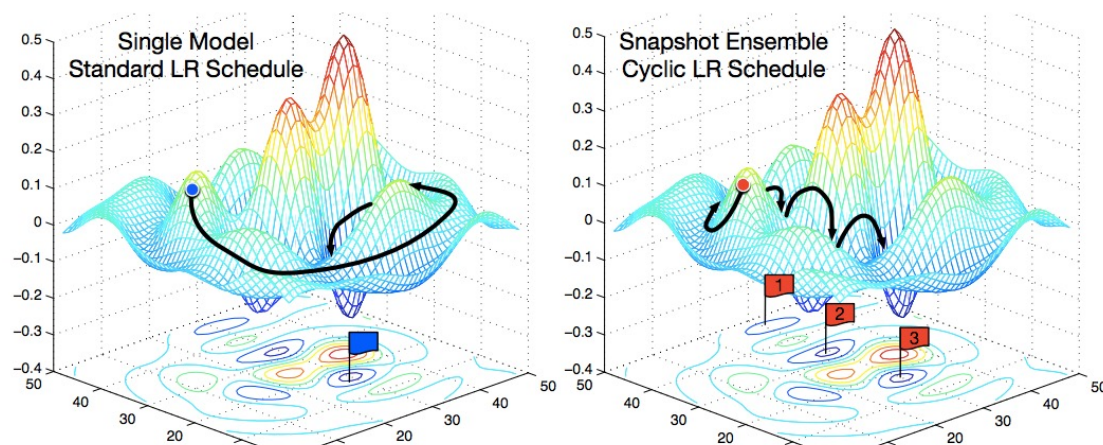
Test time

- Average predictions across multiple crops of test image
 - Sliding window prediction is a more elegant way to do this



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)



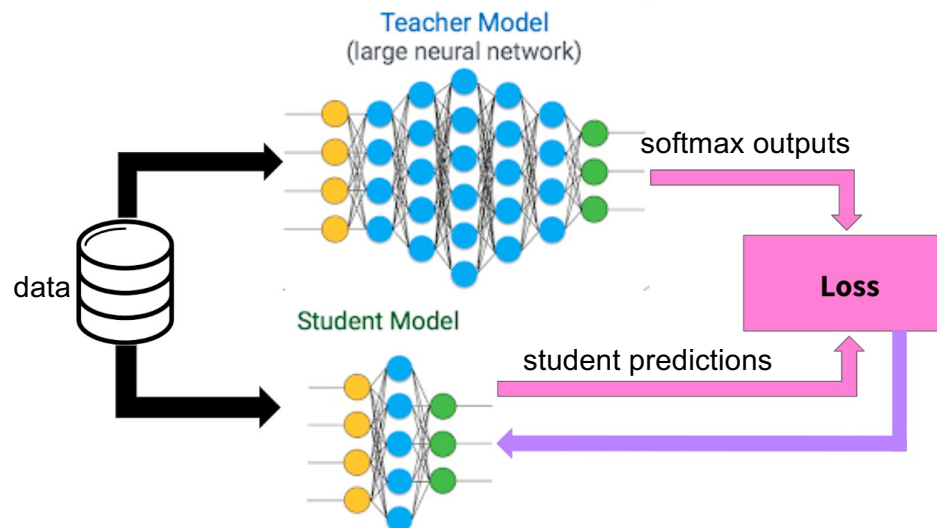
G. Huang et al., [Snapshot ensembles: Train 1, get M for free](#), ICLR 2017

Outline

- Optimization
 - Mini-batch SGD
 - Learning rate decay
 - Diagnosing learning curves
 - Adaptive methods
- Massaging the numbers
 - Data augmentation
 - Data preprocessing
 - Weight initialization
 - Batch normalization
- Regularization
- Test time: ensembles, averaging predictions
- Student-teacher training: distillation

Distillation

1. Train a *teacher* network on initial labeled dataset
2. Save the softmax outputs the teacher network for each training example
3. Train a *student* network with cross-entropy loss using the softmax outputs of the teacher network as targets



[Image source](#)

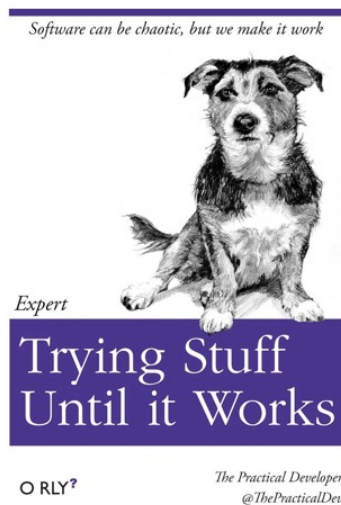
G. Hinton, O. Vinyals, J. Dean. [Distilling the knowledge in a neural network](#). arXiv 2015

Distillation

1. Train a *teacher* network on initial labeled dataset
 2. Save the softmax outputs the teacher network for each training example
 3. Train a *student* network with cross-entropy loss using the softmax outputs of the teacher network as targets
- Many uses:
 - Compressing a larger model (or even an ensemble) into a smaller one
 - “Copying” a black-box teacher model (e.g., network you can only access via an API)
 - Extending a network to additional tasks without “forgetting” old tasks ([Li and Hoiem, 2017](#))

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](#)

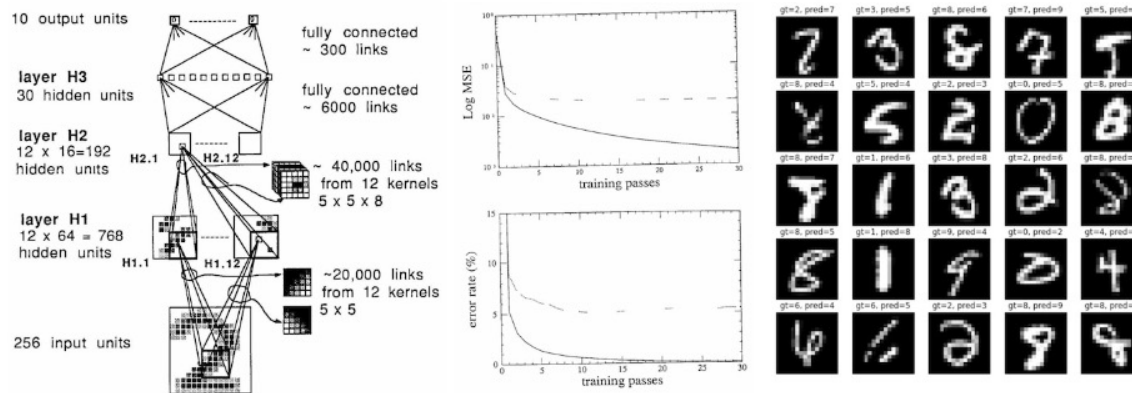


More fun reading

Deep Neural Nets: 33 years ago and 33 years from now

Mar 14, 2022

The Yann LeCun et al. (1989) paper [Backpropagation Applied to Handwritten Zip Code Recognition](#) is I believe of some historical significance because it is, to my knowledge, the earliest real-world application of a neural net trained end-to-end with backpropagation. Except for the tiny dataset (7291 16x16 grayscale images of digits) and the tiny neural network used (only 1,000 neurons), this paper reads remarkably modern today, 33 years later - it lays out a dataset, describes the neural net architecture, loss function, optimization, and reports the experimental classification error rates over training and test sets. It's all very recognizable and type checks as a modern deep learning paper, except it is from 33 years ago. So I set out to reproduce the paper 1) for fun, but 2) to use the exercise as a case study on the nature of progress in deep learning.



<http://karpathy.github.io/2022/03/14/lecun1989/>

Even more food for thought



Jason Wei 
@_jasonwei



An incredible skill that I have witnessed, especially at OpenAI, is the ability to make “yolo runs” work.

The traditional advice in academic research is, “change one thing at a time.” This approach forces you to understand the effect of each component in your model, and therefore is a reliable way to make something work. I personally do this quite religiously. However, the downside is that it takes a long time, especially if you want to understand the interactive effects among components.

A “yolo run” directly implements an ambitious new model without extensively de-risking individual components. The researcher doing the yolo run relies primarily on intuition to set hyperparameter values, decide what parts of the model matter, and anticipate potential problems. These choices are non-obvious to everyone else on the team.

Yolo runs are hard to get right because many things have to go correctly for it to work, and even a single bad hyperparameter can cause your run to fail. It is probabilistically unlikely to guess most or all of them correctly.

Yet multiple times I have seen someone make a yolo run work on the first or second try, resulting in a SOTA model. Such yolo runs are very impactful, as they can leapfrog the team forward when everyone else is stuck.

I do not know how these researchers do it; my best guess is intuition built up from decades of running experiments, a deep understanding of what matters to make a language model successful, and maybe a little bit of divine benevolence. But what I do know is that the people who can do this are surely 10-100x AI researchers. They should be given as many GPUs as they want and be protected like unicorns.

1:25 PM · Feb 13, 2024 · **475.5K** Views

196 Reposts **98** Quotes **2,244** Likes **1,012** Bookmarks

https://twitter.com/_jasonwei/status/1757486124082303073