Neural network training: The basics and beyond

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
- Transfer learning, distillation

Mini-batch SGD

Caspar David Friedrich, *Wanderer above a sea of fog*, 1817

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:

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

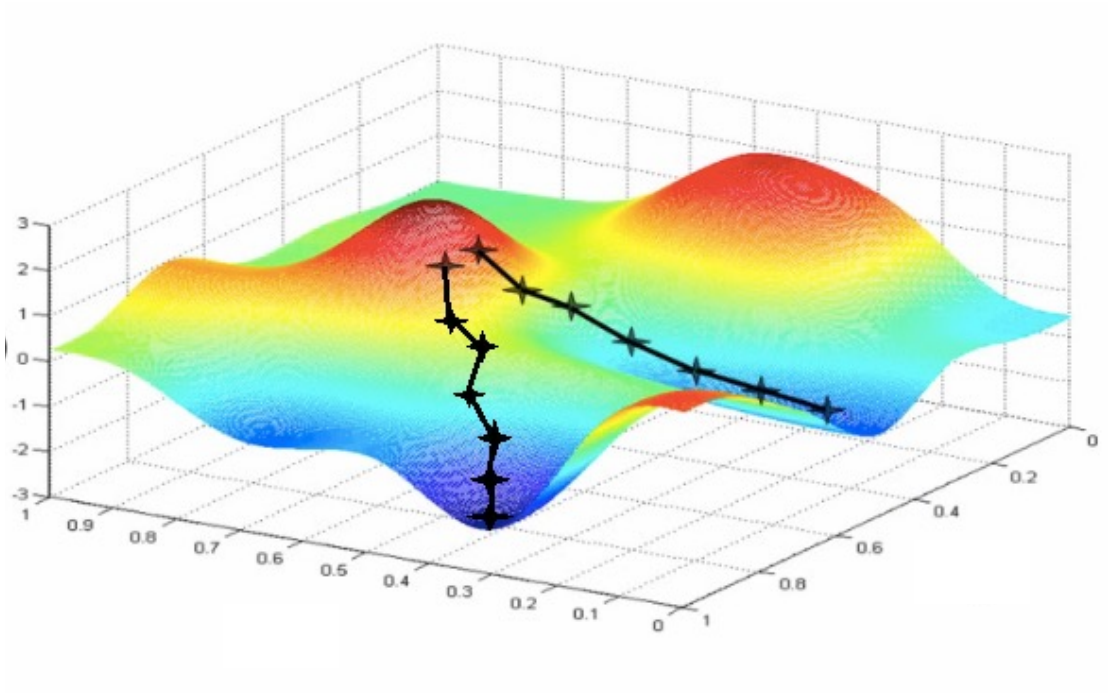
  - Check for convergence, decide whether to decay learning rate

- 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)

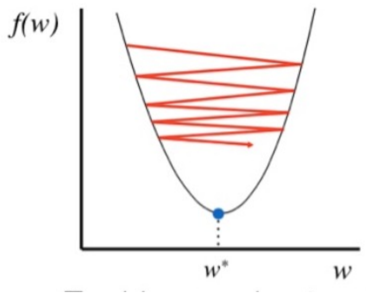
Setting the learning rate

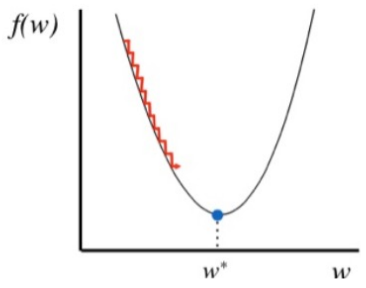


Setting the learning rate

Too high

Too low





Want: good *decay schedule*

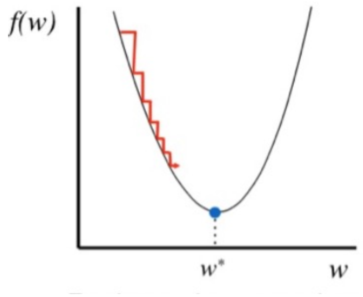
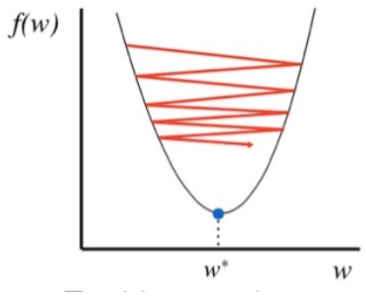


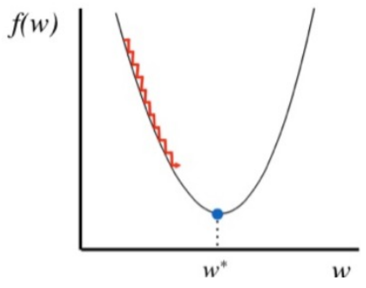
Figure source

Setting the learning rate

Too high



Too low



Want: good *decay schedule*

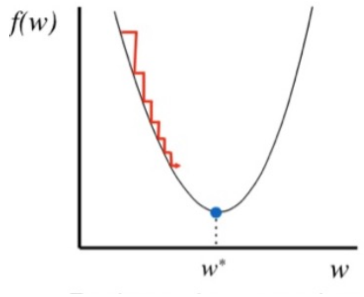
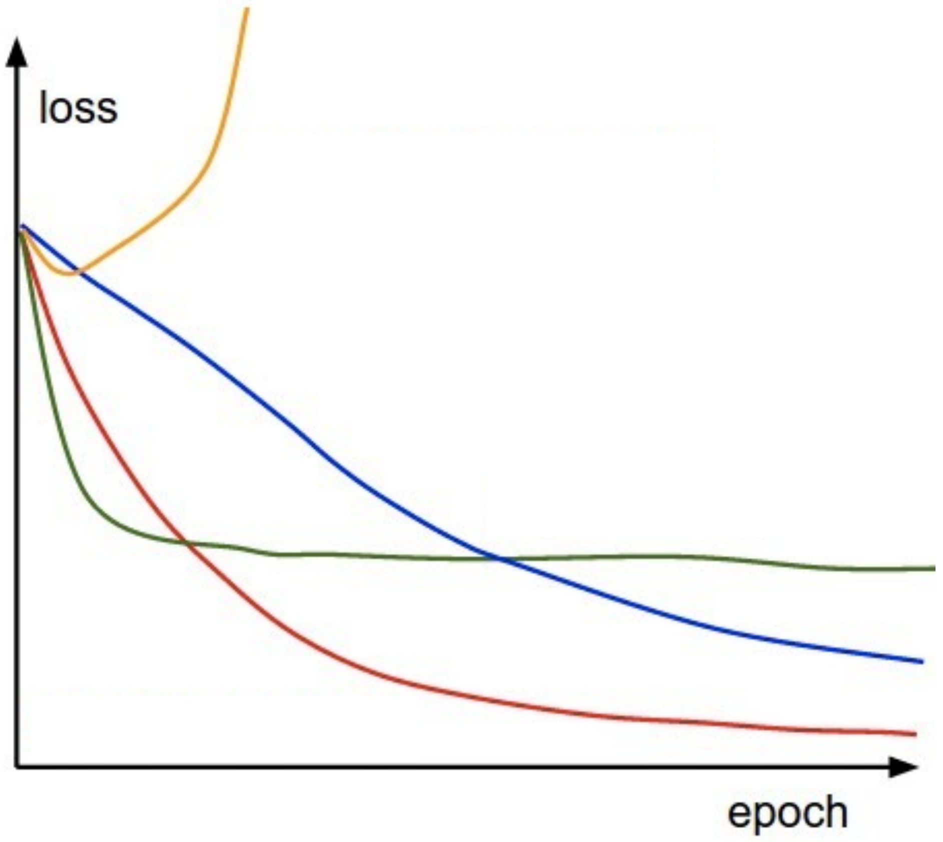


Figure source



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

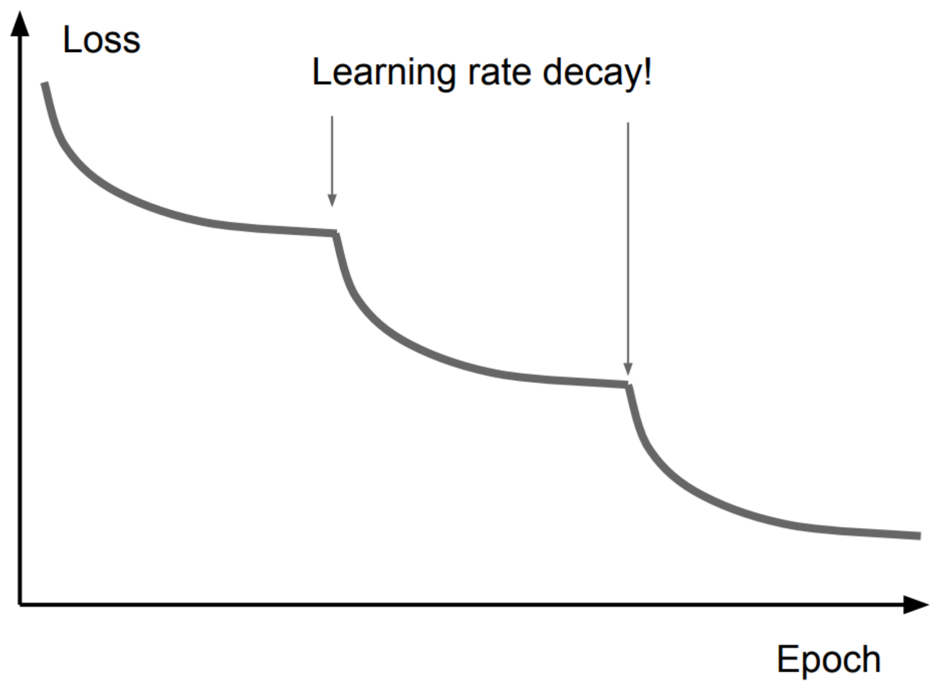


Image source: Stanford CS231n

Possible explanation

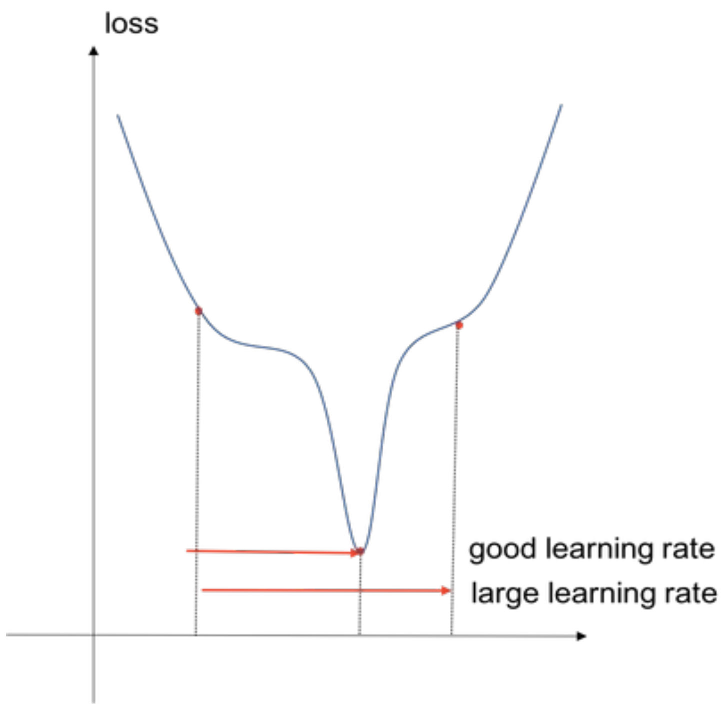
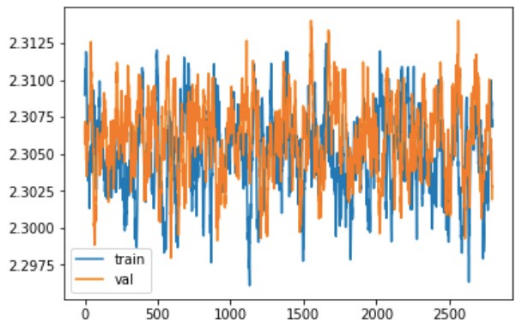


Image source

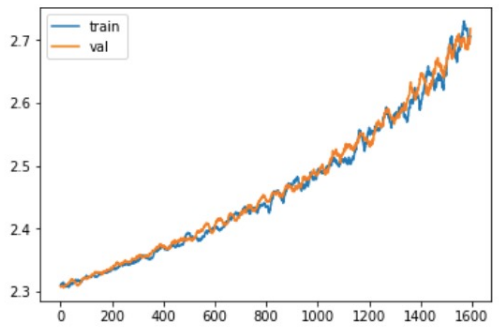
Learning rate decay

- 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)

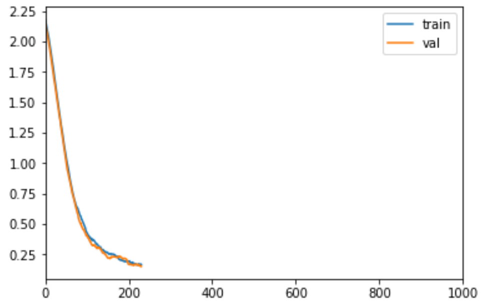
Diagnosing learning curves: Obvious problems



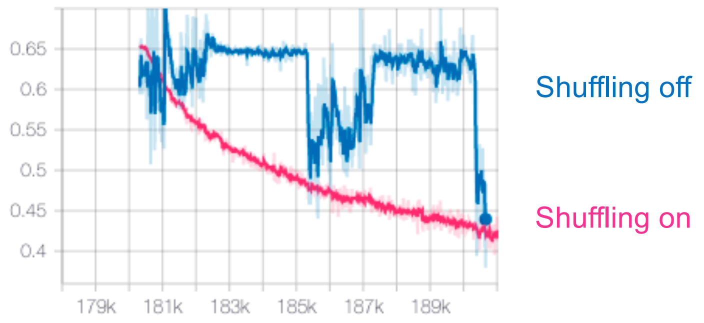
Not training
Bug in update calculation?



Error increasing
Bug in update calculation?



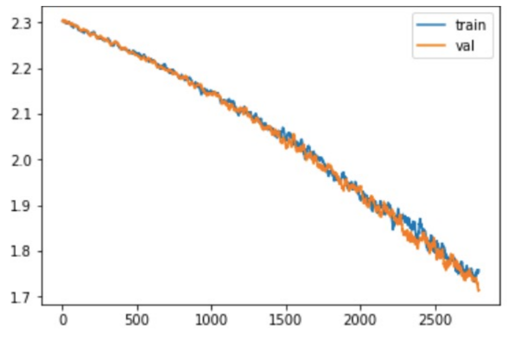
Get NaNs in the loss after a number of iterations:
Numerical instability



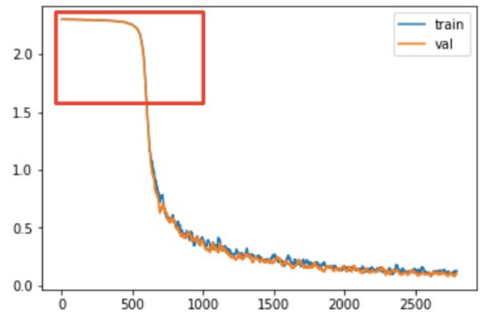
Weird cyclical patterns in loss:
Data not shuffled

Source: [Stanford CS231n]()

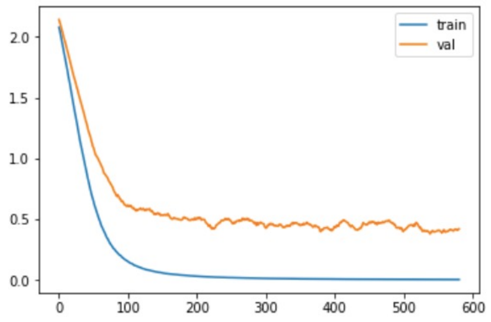
Diagnosing learning curves: Subtler behaviors



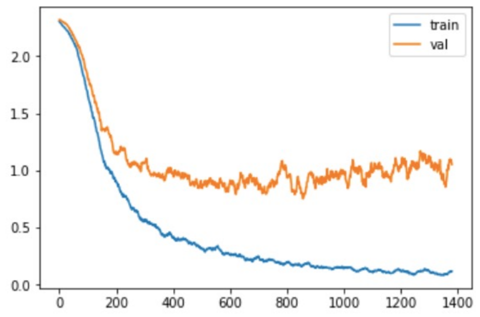
Not converged yet
Keep training, possibly increase learning rate



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

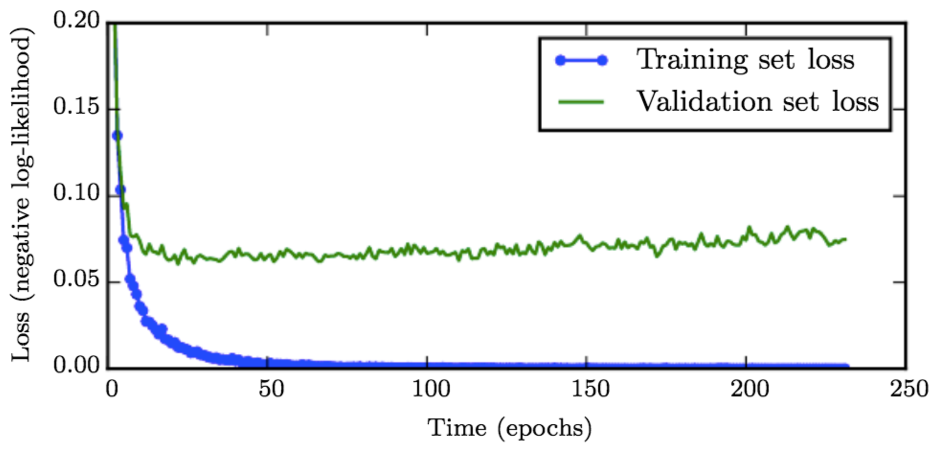
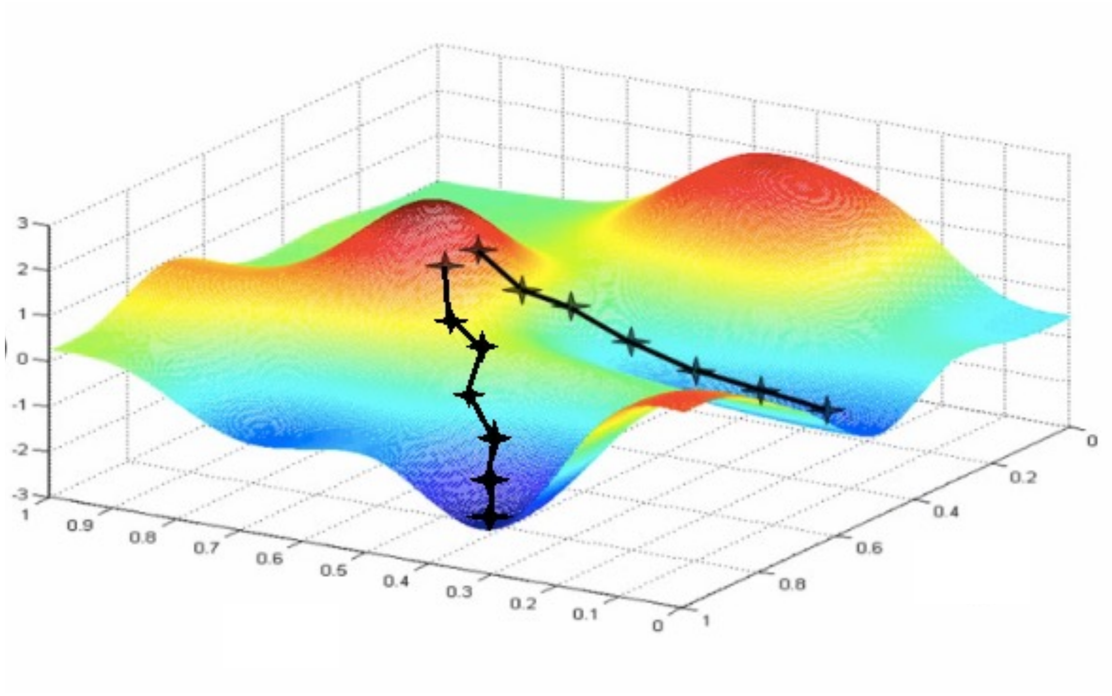


Figure from Deep Learning Book

Outline

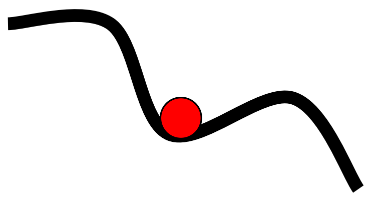
- Optimization
  - Mini-batch SGD
  - Learning rate decay
  - Diagnosing learning curves
  - Adaptive optimization methods: SGD with momentum, RMSProp, Adam

Where does SGD run into trouble?

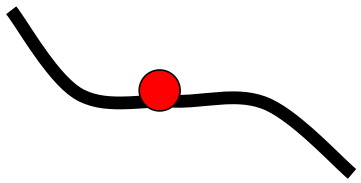


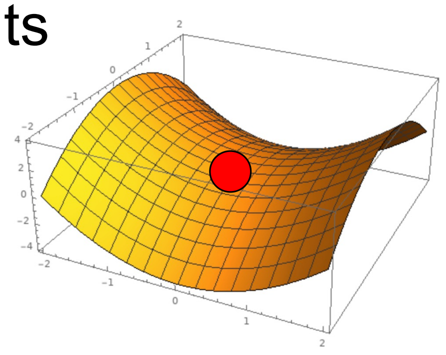
Where does SGD run into trouble?

Local minima

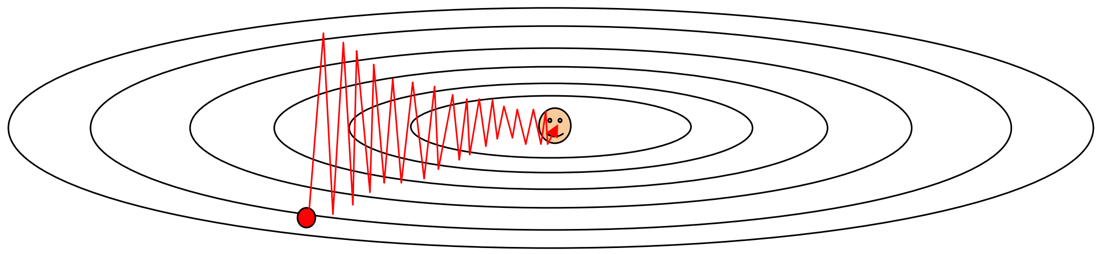


Saddle points





Poor conditioning

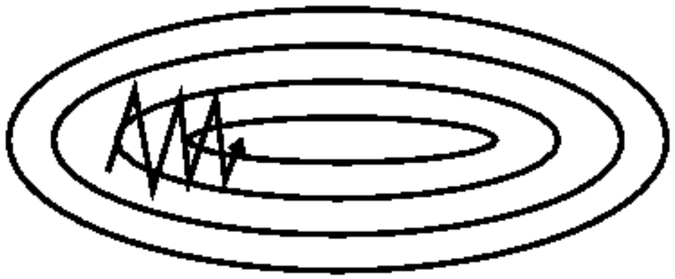


Source: J. Johnson

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

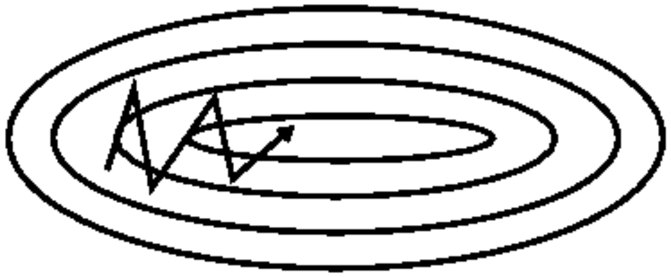


Image source

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

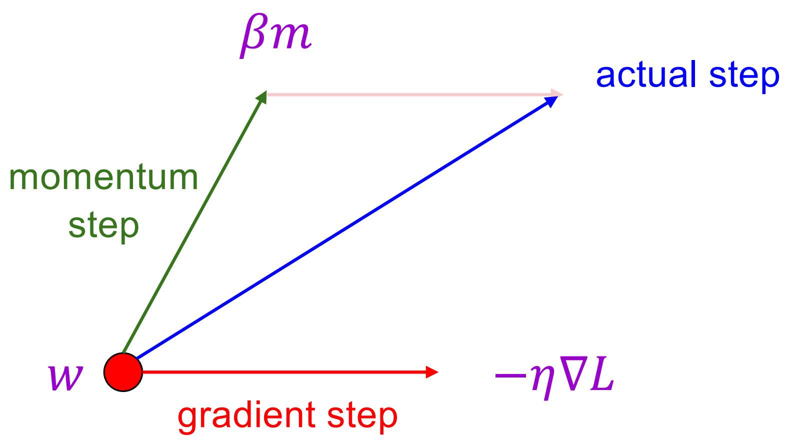


Image source

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

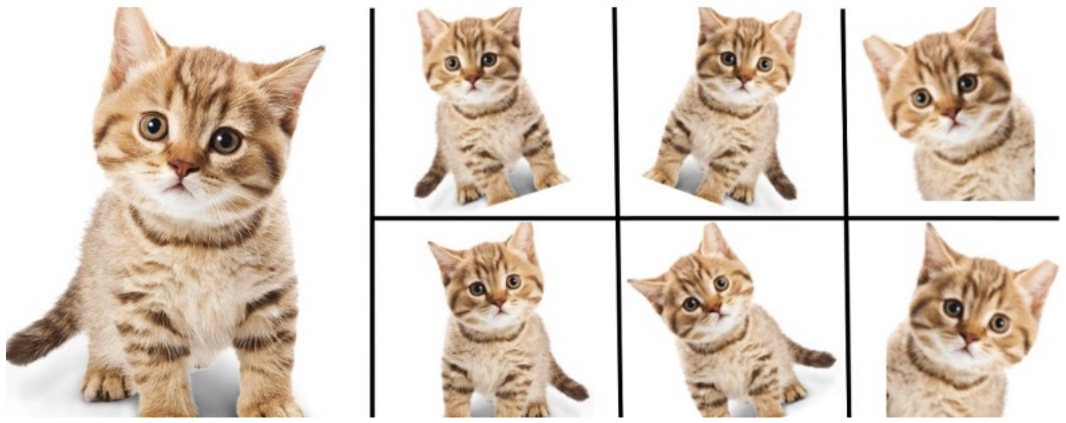


Image source

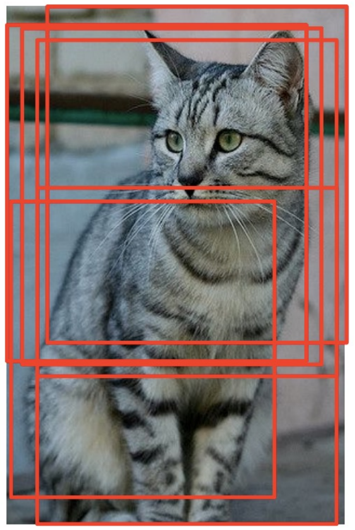


Image source

Data augmentation

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

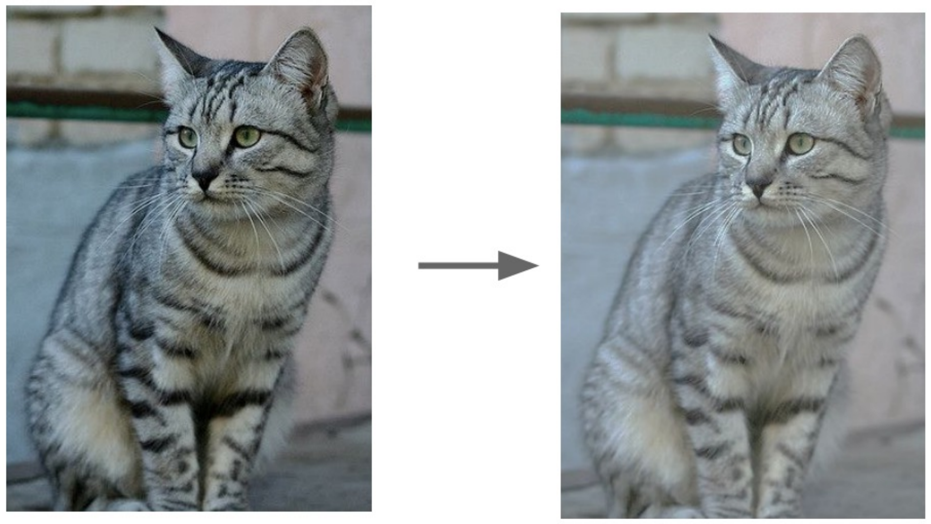


Image source

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.
    - Automatic augmentation strategies: AutoAugment, RandAugment

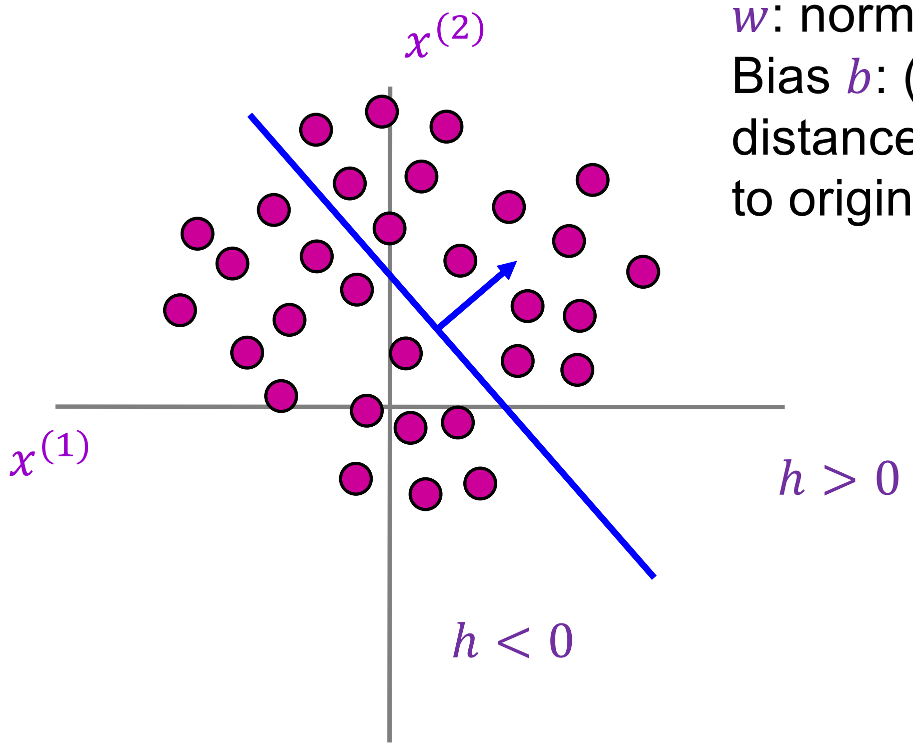
Data preprocessing

- 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

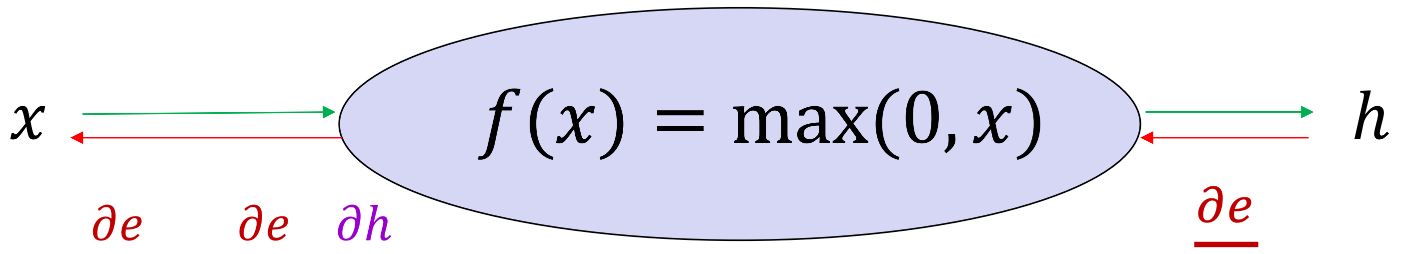
The importance of preprocessing and initialization

- 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



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

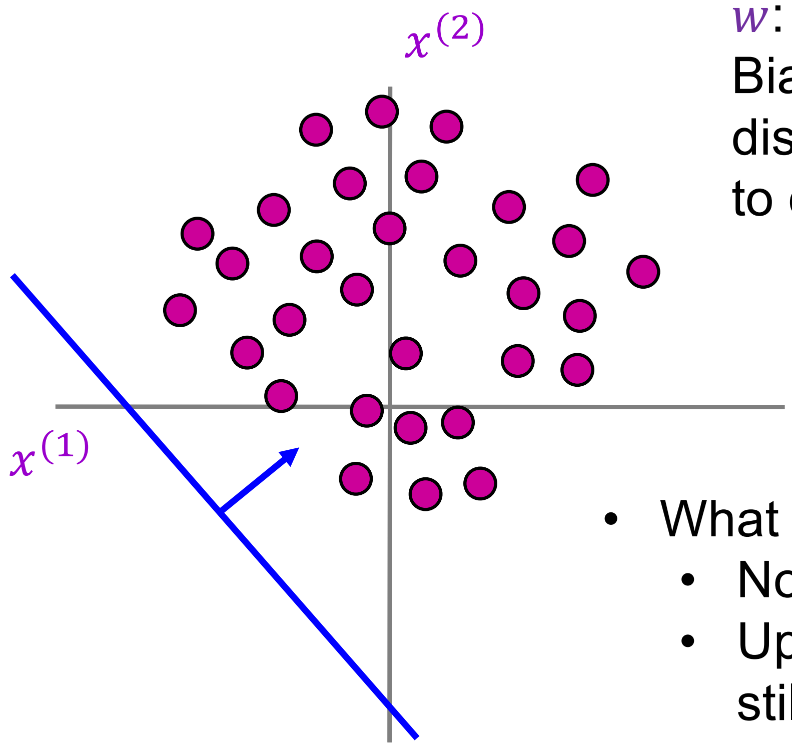
$$\frac{\partial e}{\partial h}$$

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

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

The importance of preprocessing and initialization

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

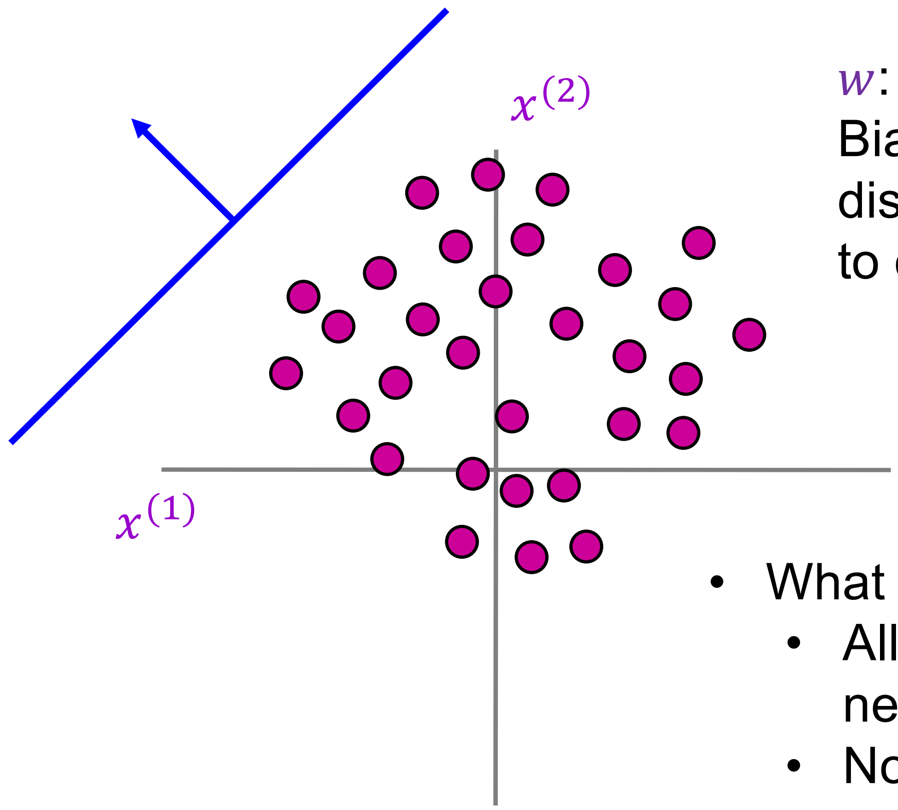


w: normal to a hyperplane
Bias b: (unnormalized) distance from hyperplane to origin

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

The importance of preprocessing and initialization

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

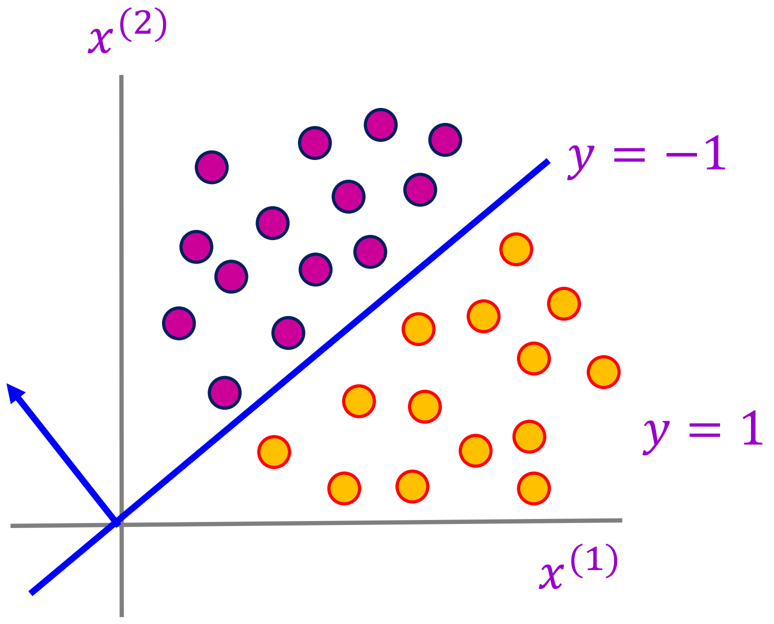


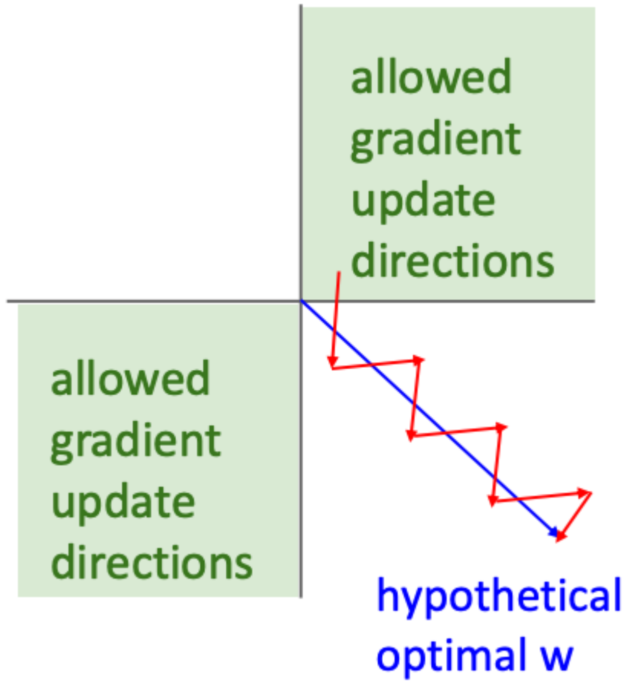
w: normal to a hyperplane
Bias b: (unnormalized) distance from hyperplane to origin

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

The importance of preprocessing and initialization

- Suppose all data is positive
- Linear perceptron with $b = 0$, initially all points are misclassified
- Recall the perceptron update: $w \leftarrow w + \eta y_i x_i$
  - Updates are all positive or all negative along individual dimensions!





allowed gradient update directions
allowed gradient update directions
hypothetical optimal w

Source: J. Johnson

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

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: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

Why?

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

Batch normalization

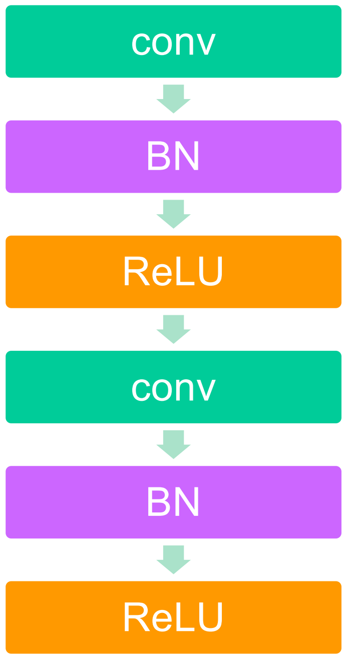
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

    $\xcancel{\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2}$    // ~~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
```

At test time (usually):

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)



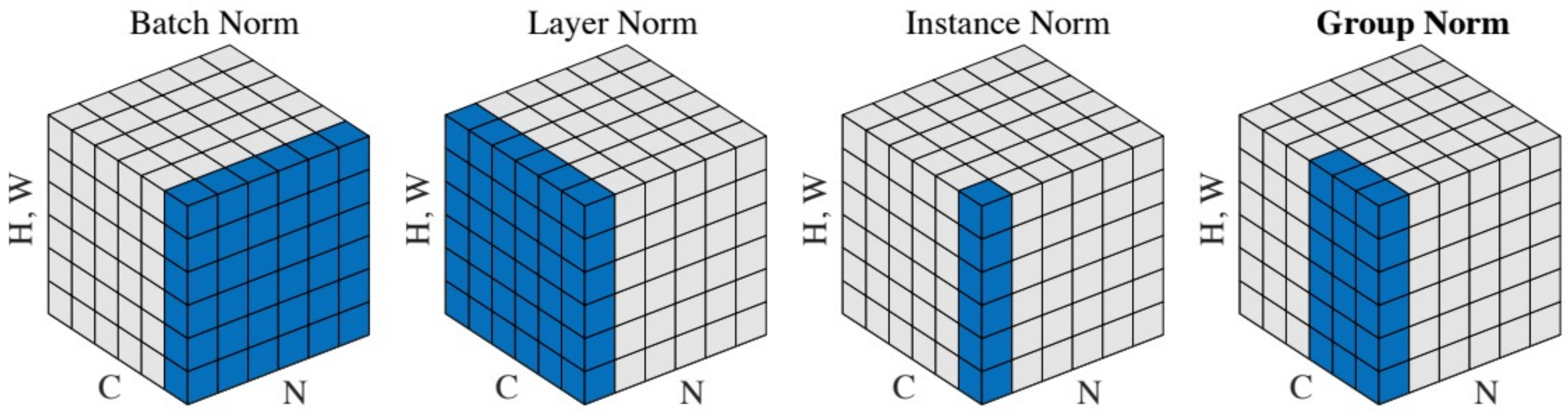
S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

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)

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)



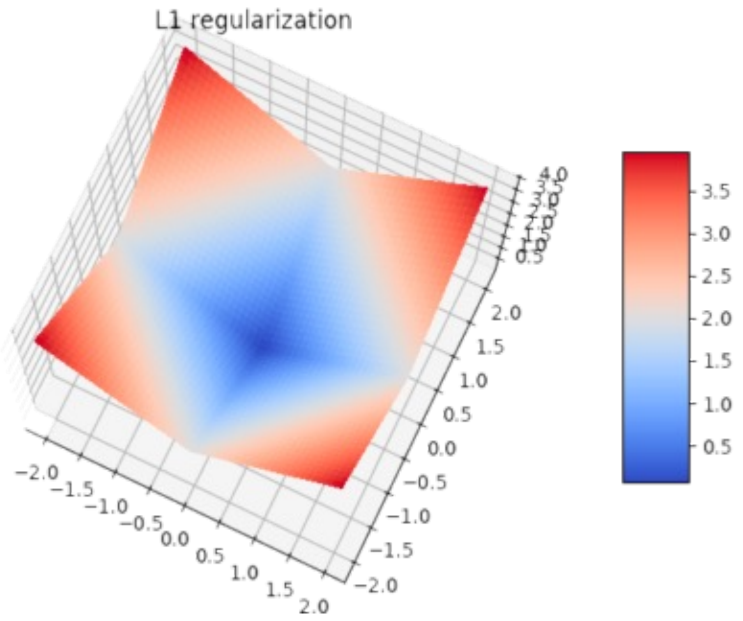
Y. Wu and K. He, Group Normalization, ECCV 2018

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



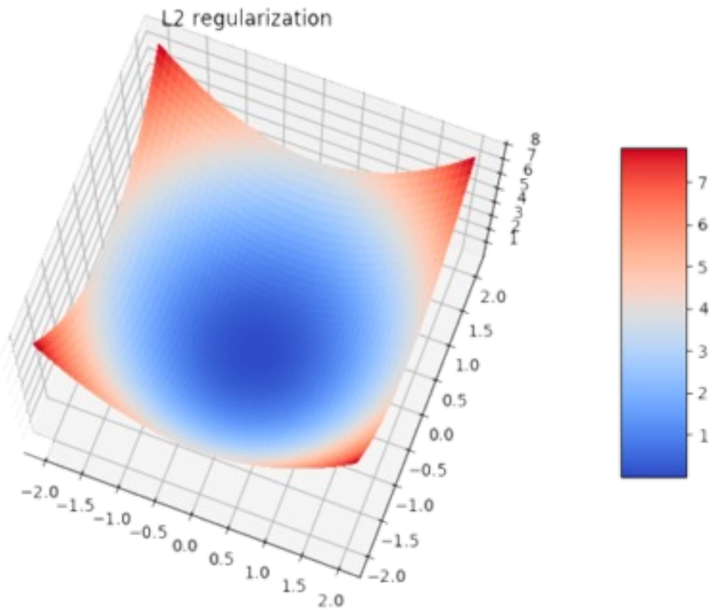


Image source

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

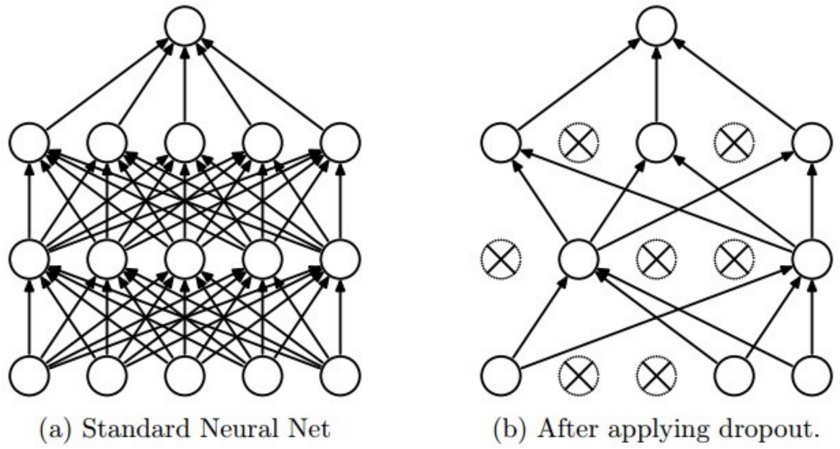
Adapted from J. Johnson

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
  - Recall: when using softmax loss, replace hard 1 and 0 prediction targets with "soft" targets of $1-\epsilon$ and $\frac{\epsilon}{C-1}$

Dropout

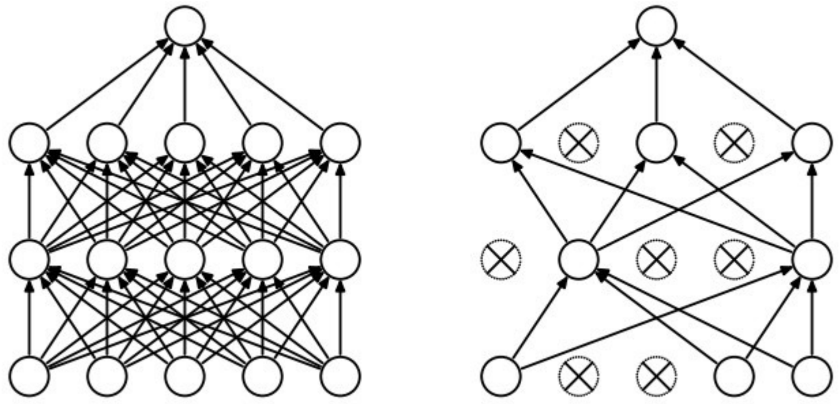
- At training time, in each forward pass, turn off some neurons with probability p
- At test time, to have deterministic behavior, multiply output of neuron by p



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

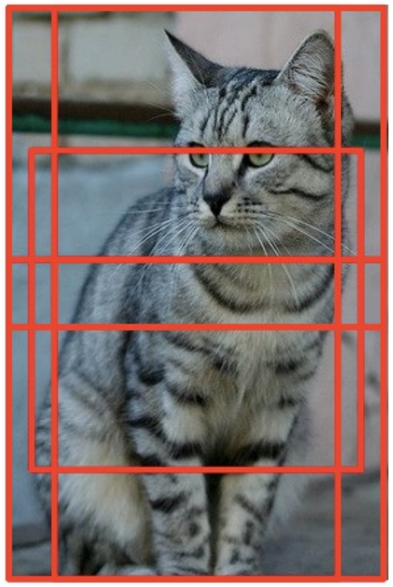
- Against
  - Slows down convergence
  - Made redundant by batch normalization or possibly even clashes with it
  - Unnecessary for larger datasets or with sufficient data augmentation
- In favor
  - Can still help for certain models and in certain situations: e.g., used in Wide Residual Networks

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

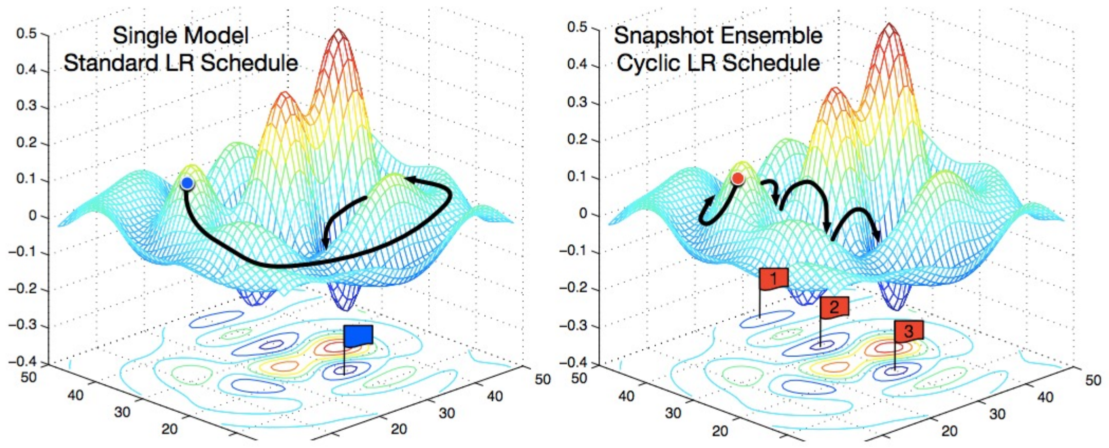
Test time

- 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)

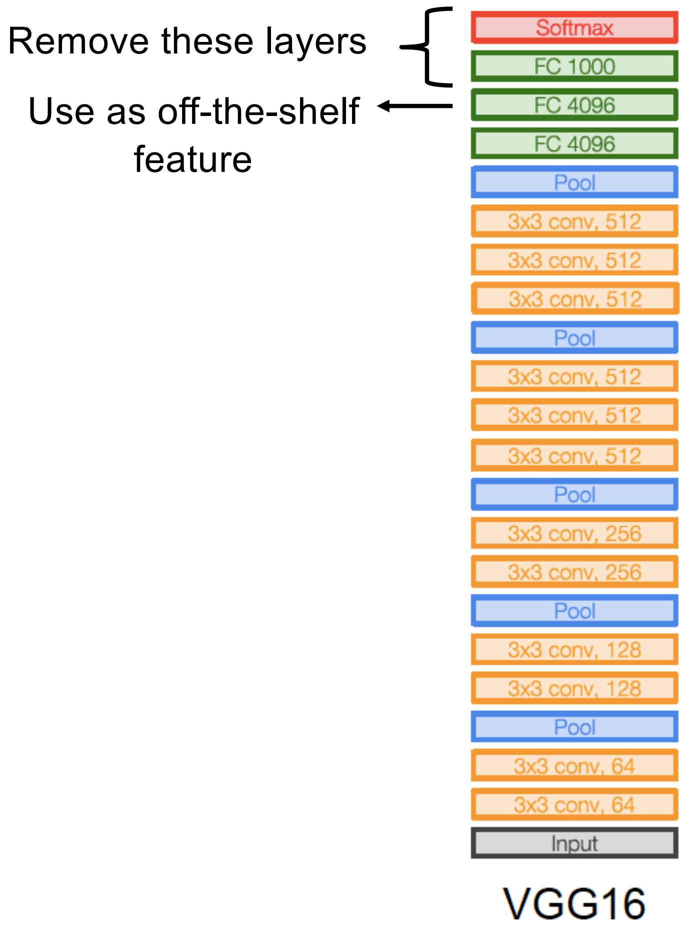


G. Huang et al., Snapshot ensembles: Train 1, get M for free, ICLR 2017

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: ensembles, averaging predictions
- Transfer learning, distillation

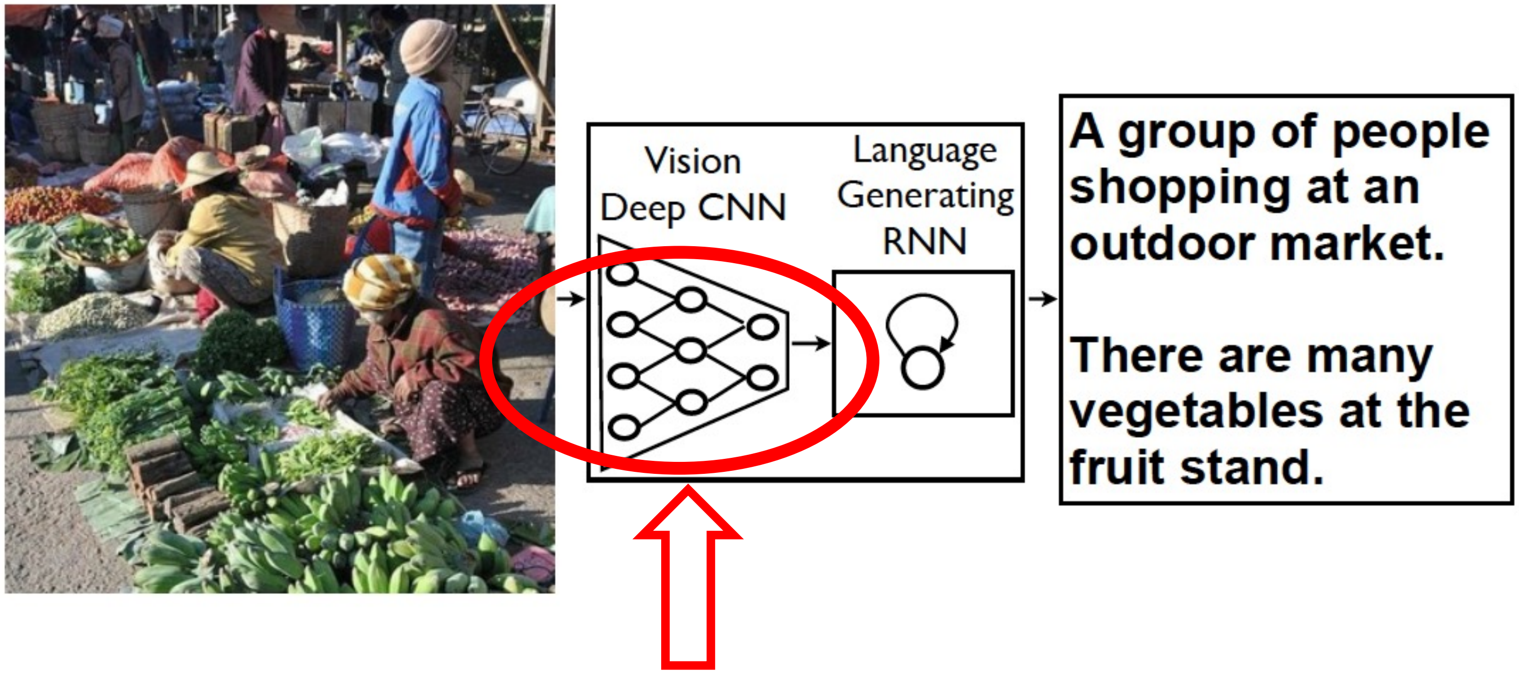
How to use a pre-trained network for a new task?



- Strategy 1: Use as feature extractor

A. Razavian et al. CNN Features Off-the-Shelf: An Astounding Baseline for Recognition. CVPR workshops, 2014

Example: CNNs for image captioning



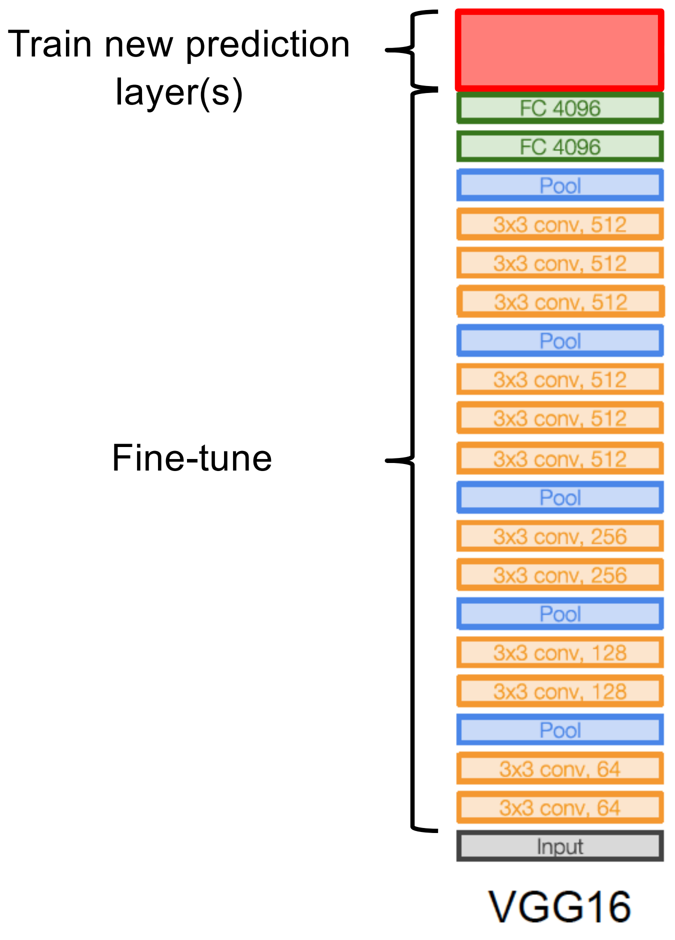
A group of people shopping at an outdoor market.

There are many vegetables at the fruit stand.

FC vectors from pre-trained network

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. [Show and tell: A neural image caption generator](). CVPR 2015

How to use a pre-trained network for a new task?



- Strategy 2: Transfer learning

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

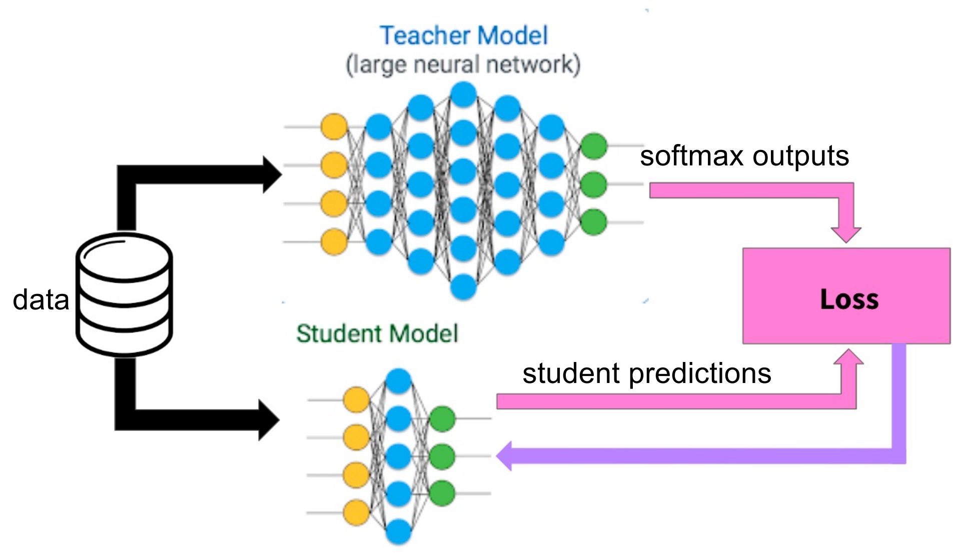


Image source

G. Hinton, O. Vinyals, J. Dean. Distilling the knowledge in a neural network. arXiv 2015

Distillation

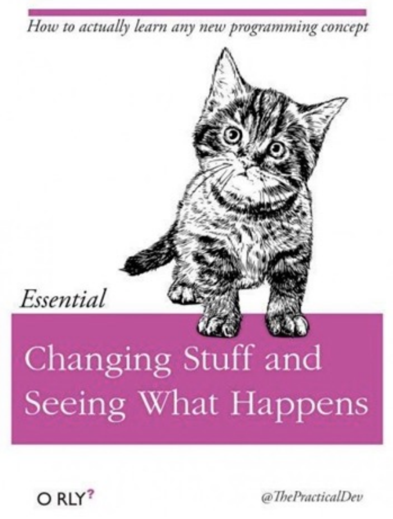
1. Train a *teacher* network on initial labeled dataset
2. Save the softmax outputs the teacher network for each training example
3. Train a *student* network with cross-entropy loss using the softmax outputs of the teacher network as targets

- 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](#)

Software can be chaotic, but we make it work
Expert
Trying Stuff Until it Works
O RLY?
The Practical Developer
@ThePracticalDev


How to actually learn any new programming concept
Essential
Changing Stuff and Seeing What Happens
O RLY?
@ThePracticalDev

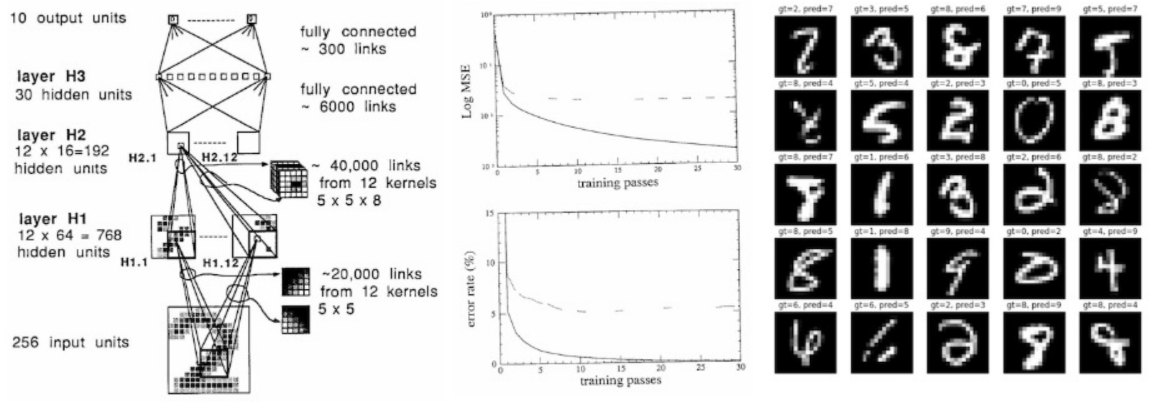
The internet will make those bad words go away
Essential
Googling the Error Message
O RLY?
The Practical Developer
@ThePracticalDev

More fun reading

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

Mar 14, 2022

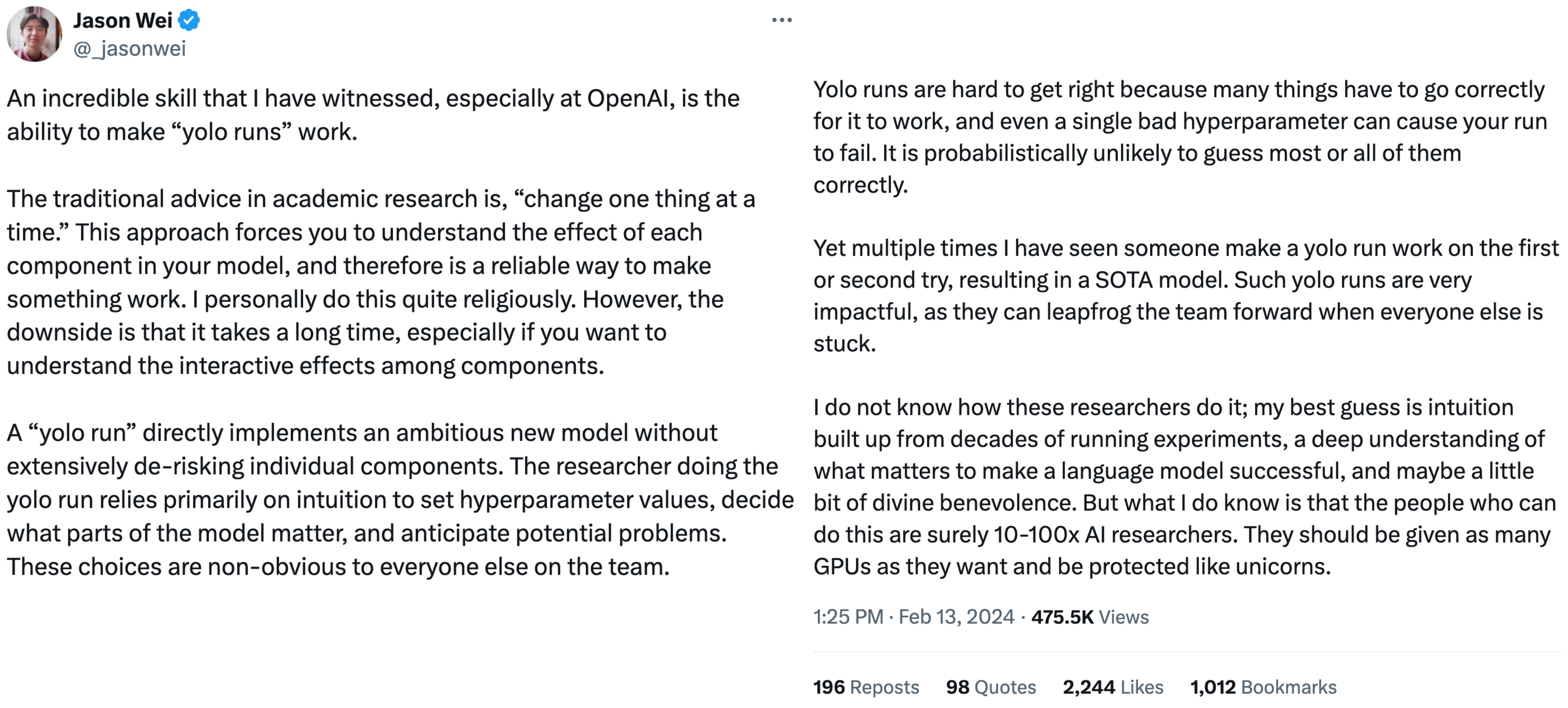
The Yann LeCun et al. (1989) paper Backpropagation Applied to Handwritten Zip Code Recognition is I believe of some historical significance because it is, to my knowledge, the earliest real-world application of a neural net trained end-to-end with backpropagation. Except for the tiny dataset (7291 16x16 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.



10 output units
fully connected ~ 300 links
layer H3 30 hidden units
fully connected ~ 6000 links
layer H2 12 x 16=192 hidden units
H2.1
H2.12
~ 40,000 links from 12 kernels 5 x 5 x 8
layer H1 12 x 64 = 768 hidden units
H1.1
H1.12
~20,000 links from 12 kernels 5 x 5
256 input units
Log MSE
training passes
error rate (%)
training passes

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