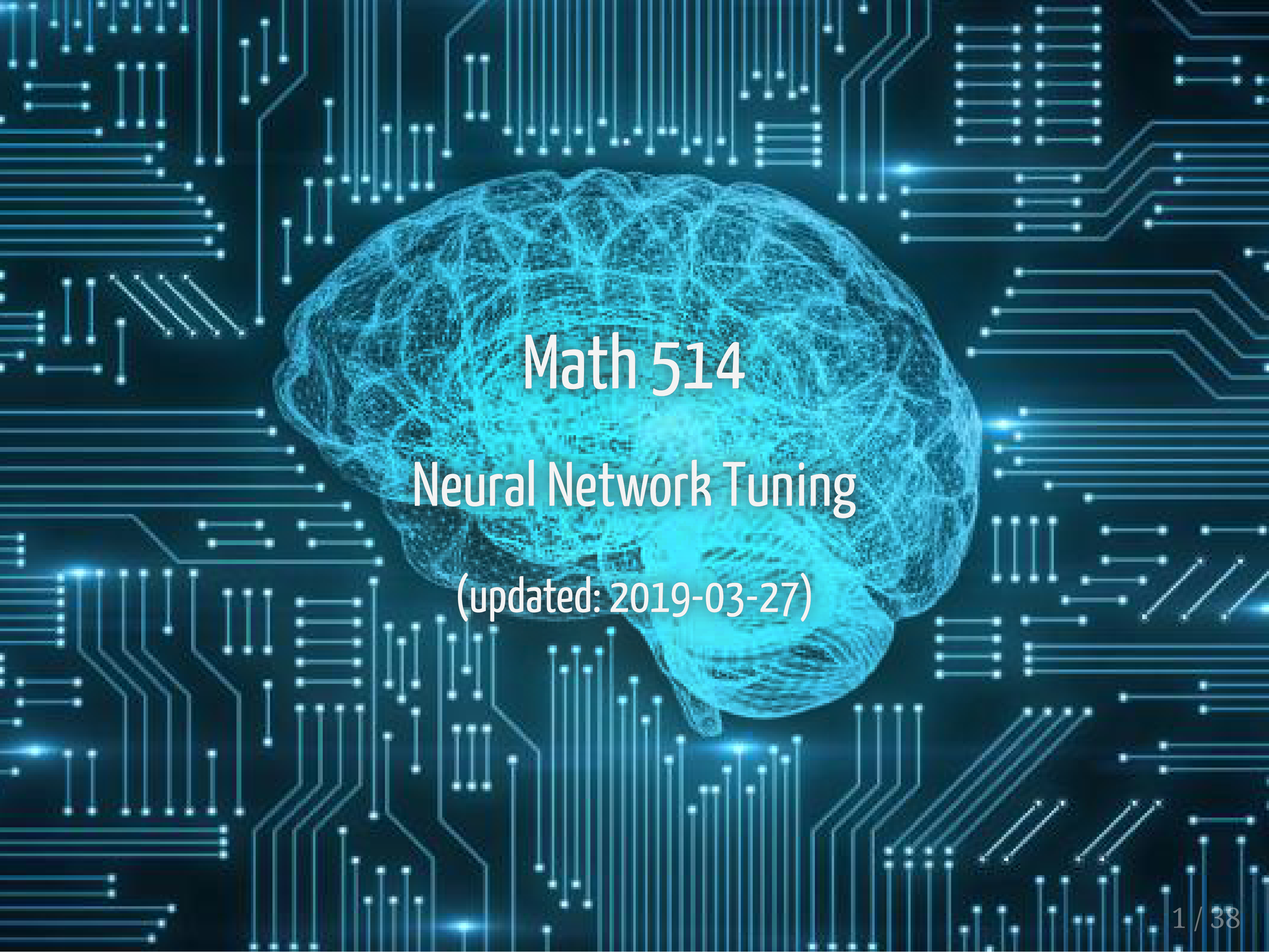
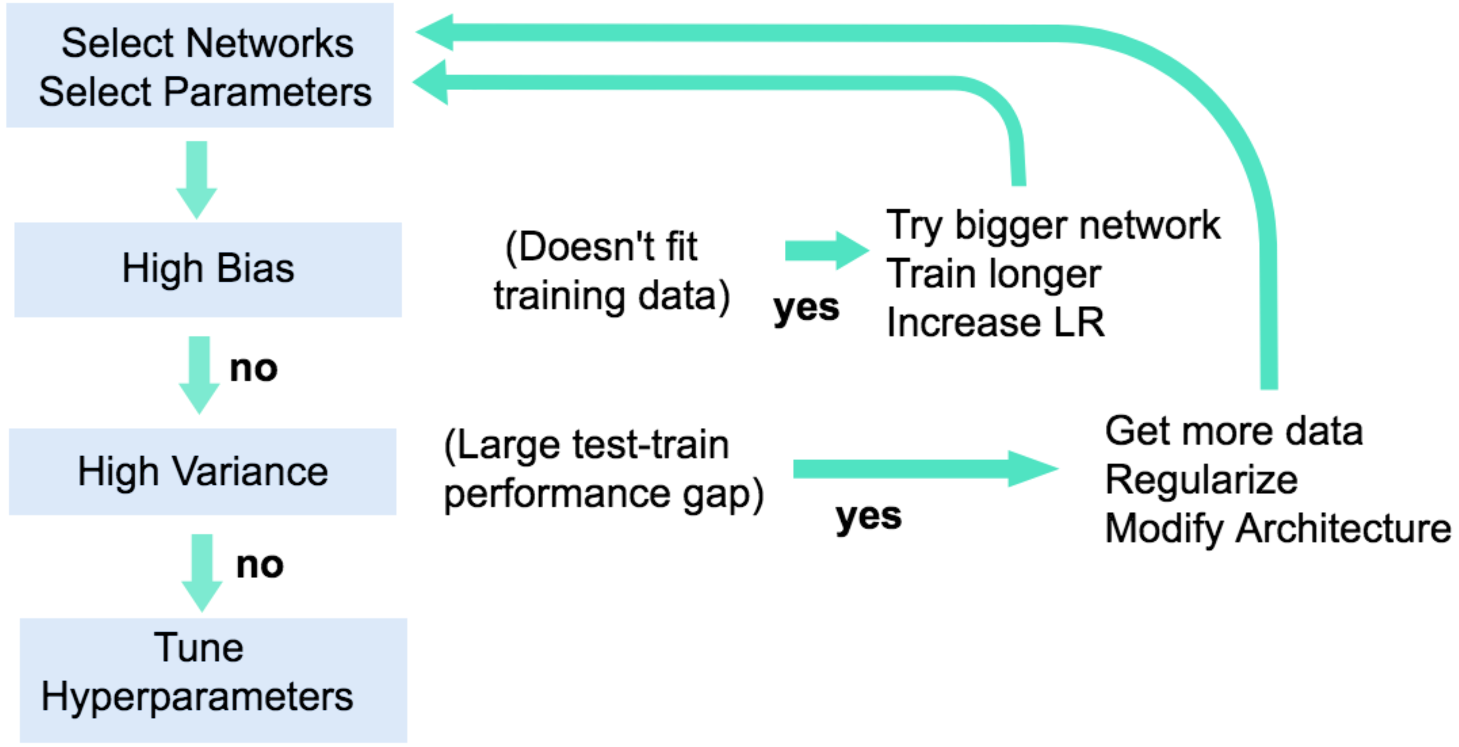


Math 514

Neural Network Tuning

(updated: 2019-03-27)

Initial Training Cycle



Training Recipe

1. Normalize inputs

- Center/Scale

2. Select network architecture

- Cost function, activation, layers, etc.

3. Initialize parameters (weights and biases)

- Covered later

4. Select regularization

- Will cover more methods today

5. Debug

6. Tune hyper-parameters

- Improve working model

7. Combine models.

Training/Tuning

Good Practices:

- Make sure test/train split is done randomly. Don't assume received data is random.

- Shuffle data when forming mini-batches (Chunker-like code).
- Use good coding practices
    - Write functions.
    - Unit test functions.
- Start with small subset of data until confident model is working.

Training

The goal of training a neural network is to minimize a cost function while maximizing the ability to *generalize*.

"Learning with deep neural networks displays good *generalization* behavior in practice, a phenomenon which remains largely unexplained. A major issue still left unresolved … is the precise relationship between optimization and *implicit generalization*."

Exploring Generalization in Deep Learning

Neyshabur, Bhojanapalli, McAllester and Srebro - 31st NIPS Conference, 2017

Improving Generalization

Dataset Augmentation

Neural networks generalize better when more training data is available. One way to increase the dataset is to generate fake data (with appropriate statistics) and add it to the training set.

- This is especially useful for image and object recognition. The dataset augmentation could be translating, rotating or scaling the input images. This often results in better generalization in these tasks.

- Augmenting the data by perturbing it results in better robustness.

- Adding noise also acts like data augmentation
    - Add noise to activations (layer outputs)
    - Add noise to weights
    - Add noise to the gradients
    - Add noise to the outputs

Improving Generalization

Transfer Learning

If data augmentation is not possible, then taking a neural network trained in one context and use it in another with little additional training.

- If the tasks are similar enough, then the features at later layers of the network ought to be good features for the new task.

- If little training data is available, but the tasks are similar, all you may need to do is to train a new linear layer at the output of the pre-trained network.

- If more data is available, it may still be a good idea to use transfer learning and tune more of the layers.

Improving Generalization

Regularization

Any modification to a learning algorithm that is intended to reduce its generalization error, but not its training error.

- Regularization is intended to counteract overfitting.
- Regularization may increase model bias (cost on training data) while reducing variance.
- Picking model size/complexity is difficult. Consider choosing a large model and regularize appropriately.

Improving Generalization

Batch normalization Recall that network input data is always normalized. One common option is z-scaling

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}(x_j^{(i)} - \mu_j)^2$$

$$\tilde{x}_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

Several Benefits:

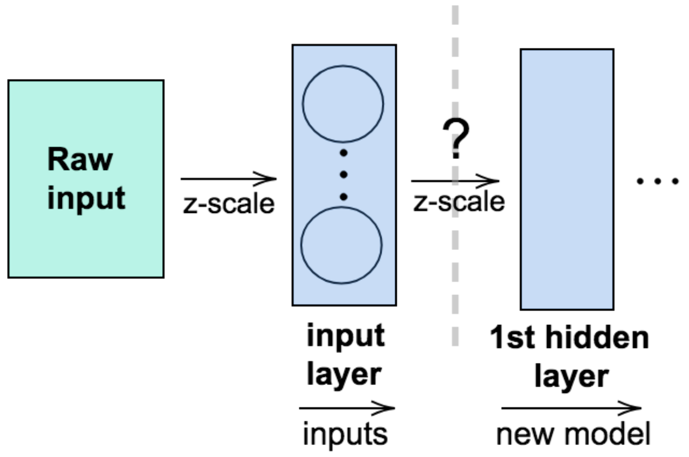
- Data components put on common scale
- Level curves become more circular leading to faster learning
- More predictable learning

Can benefits of normalization be extended to other layers in a deep network?

Improving Generalization/Batch Normalization

As learning progresses and weights are adjusted, the output distribution of a layer changes.

- This causes a changing learning environment for the subsequent layer.
- As output changes it might saturate the next layer.
- Think of each layer as learning a new representation of its input data.
- Normalizing input to each layer "decouples" learning.



Ref: Ioffe & Szegedy.

Improving Generalization/Batch Normalization

Inserting z-scaling between each layer does not work well.

- Too restrictive, means and variance can't be fixed.

Batch normalization uses a 2-step transformation:

Note: *Some researchers normalize the layer output $a^{[l]}$ instead of $z^{[l]}$*

Step 1.

The inputs $\mathbf{z}^{(i)}$ to a layer are first z-scaled

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(z_j^{(i)} - \mu_j)^2$$

$$\tilde{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j + \epsilon}}$$

- The scaling parameters μ_i, σ_j^2 are computed over the mini-batch

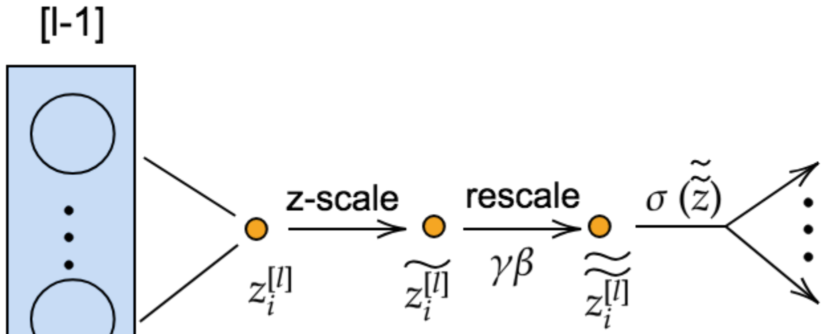
Batch Normalization

Step 2.

A second transformation with 2 new parameter vectors $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$

$$\tilde{\tilde{z}}_j^{(i)} = \boldsymbol{\gamma}_j \tilde{z}_j^{(i)} + \boldsymbol{\beta}_j$$

- Gives trainable control over layer-layer mean and variance
- Adds new parameters which will have gradients $\nabla_\gamma C$ and $\nabla_\beta C$



Note: Original distribution is restored if

$$\boldsymbol{\gamma} = \sqrt{\sigma^2 + \epsilon}$$

$$\boldsymbol{\beta} = \mu$$

Improving Generalization/Batch Normalization

$$\mathbf{z}^{[l]} = W^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$

$$\boldsymbol{\mu} = \frac{1}{m}\sum_{i=1}^{m} W^{[\ell]}\mathbf{a}^{[\ell-1](i)} + \mathbf{b}^{[\ell]}$$

When $z^{[l]} \rightarrow \tilde{z}^{[l]}$ subtracting out $\boldsymbol{\mu}$ cancels out $\mathbf{b}^{[\ell]}$

$$\tilde{\mathbf{z}}^{[l]} = \frac{(\mathbf{z}^{[l]} - \boldsymbol{\mu})}{\sqrt{\sigma^2 + \epsilon}}$$

$$\tilde{\tilde{\mathbf{z}}}^{[l]} = \boldsymbol{\gamma}^{[l]} \odot \tilde{\mathbf{z}}^{[l]} + \boldsymbol{\beta}^{[l]}$$

Improving Generalization/Batch Normalization

Parameter space has expanded

Weights:

$$W^{[1]}, W^{[2]}, \cdots, W^{[l]}$$

means:

$$\boldsymbol{\beta}^{[1]}, \boldsymbol{\beta}^{[2]}, \cdots, \boldsymbol{\beta}^{[l]}$$

variances:

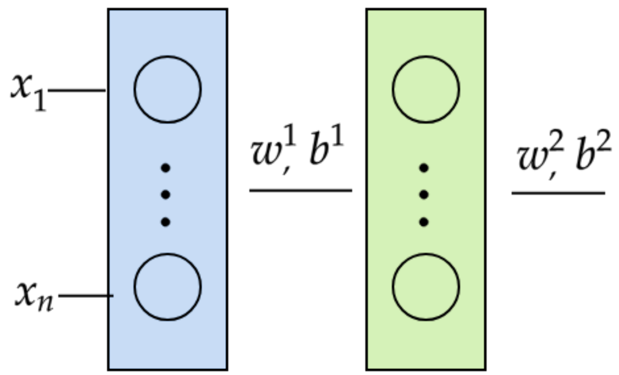
$$\boldsymbol{\gamma}^{[1]}, \boldsymbol{\gamma}^{[2]}, \cdots, \boldsymbol{\gamma}^{[l]}$$

- Will not derive gradients/backpropogation for batch normalization.
- Frameworks typically treat batch normalization as additional layers.

Batch Normalization

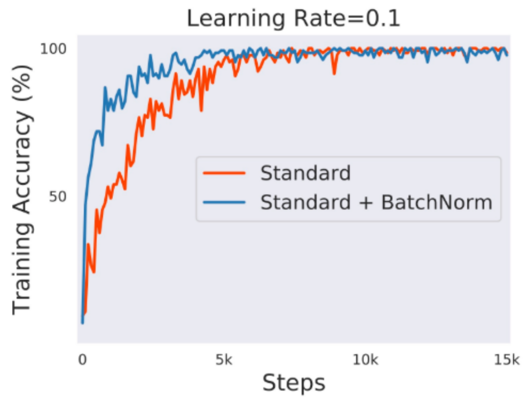
Intuition

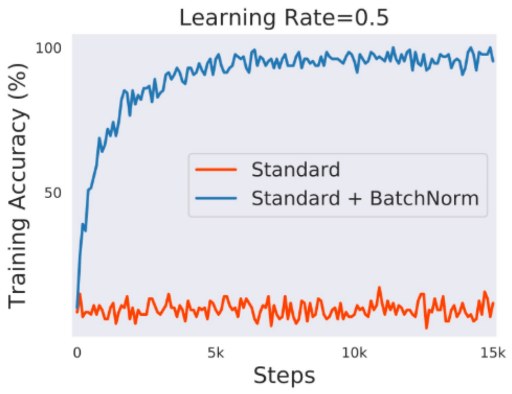
- As model learns, $W^{[1]}, \mathbf{b}^{[1]}$ evolve and change $\mathbf{a}^{[1]}$ distribution sent to layer[2].
- This can slow down learning in layer [2].
- It's a valuable insight to look at a deep network as a sequence of networks.
- Batch normalization scales using mini-batches. This adds noise to layer outputs and has a regularization effect.



Why Does Batch Normalization Work?

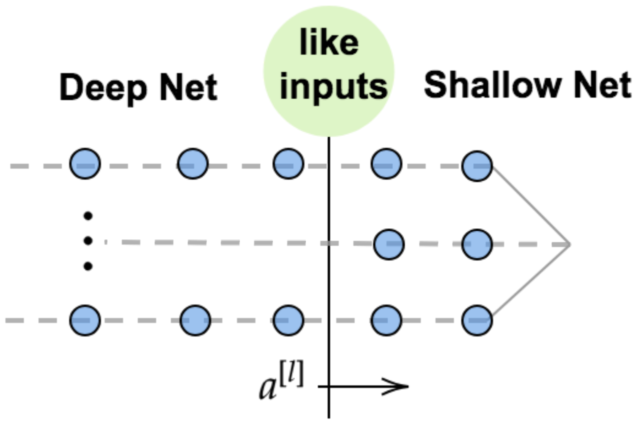
1. Intuition: Normalizing input speeds convergence
   - Batch normalization does same at hidden layers.
2. Makes deep layer weights more robust to layer-one changes. (decouples)





Data distribution changes: covariate shift

Why Does Batch Normalization Work?



If we were training right shallow net, would normalize inputs.

- With BN, inputs to each layer have mean β, variance γ
  - stable "environment".
- Without batch normalization distribution of $\mathbf{a}^{[\ell]}$ continually changes
  - Covariate shift
- As earlier layers adapt, later layers not thrown off
- Layers independently adapt
  - Weaker layer-layer coupling

Batch Normalization

Mini-batches are a training time concept.

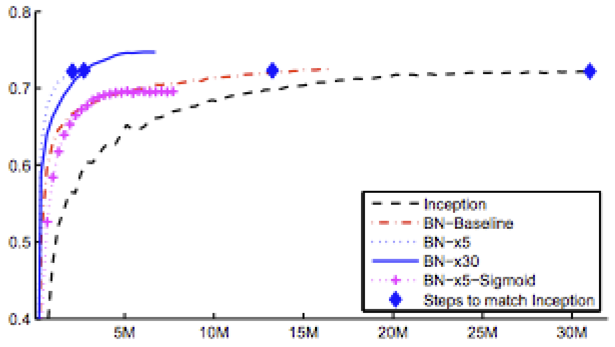
To evaluate model at test time, need values for $\boldsymbol{\mu}^{[\ell]}$ and $\boldsymbol{\sigma}^{\mathbf{2}[\boldsymbol{\ell}]}$.

- During training, exponentially weighted averages (EMA) of mini-batch $\boldsymbol{\mu}^{[\ell]}$ and $\boldsymbol{\sigma}^{\mathbf{2}[\ell]}$ values are computed.
- EMA $\boldsymbol{\gamma}$ and $\boldsymbol{\sigma}^{\mathbf{2}}$ values used at test time.

Batch Normalization

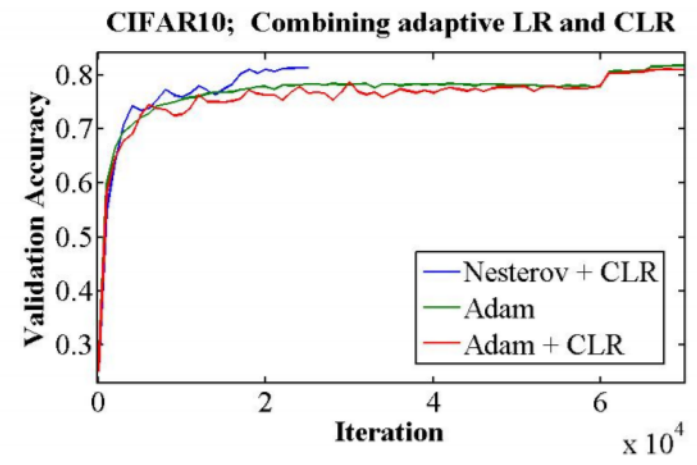
"Batch Normalization: Accelerating Deep Network Training by Reducing Covariate Shift," Ioffe, Szegedy 2015.

- Increase learning rate.
- Remove dropout.
- Reduce L_2 regularization.
- Accelerate LR decay.
- Shuffle training more thoroughly.
- Reduce photometric distortions.



Cyclic Rates for Training

Cyclic Rates for Training Neural Networks, Smith, 2017



A short run of only a few epochs where the learning rate linearly increases is sufficient to estimate boundary learning rates for the CLR policies.

Cyclical learning rate (CLR) method practically eliminates the need to tune the learning rate yet achieve near optimal classification accuracy

This policy is easy to implement and unlike adaptive learning rate methods, incurs essentially no additional computational expense.

Hyper-Parameter Tuning

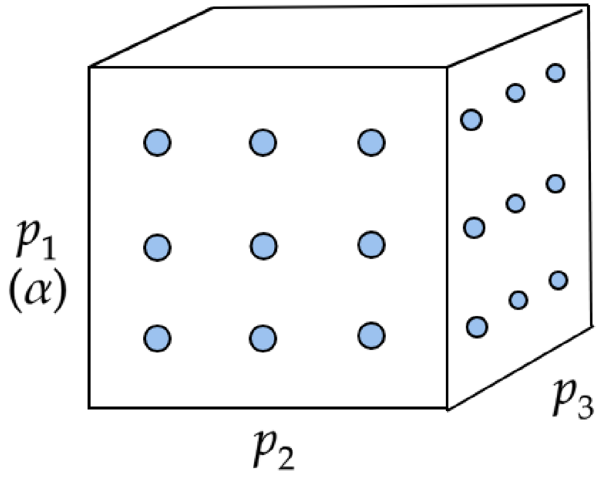
Grid search not recommended

Parameters are adjusted using gradient descent. Hyper-parameters are determined by repeated training trials.

Learning rate:

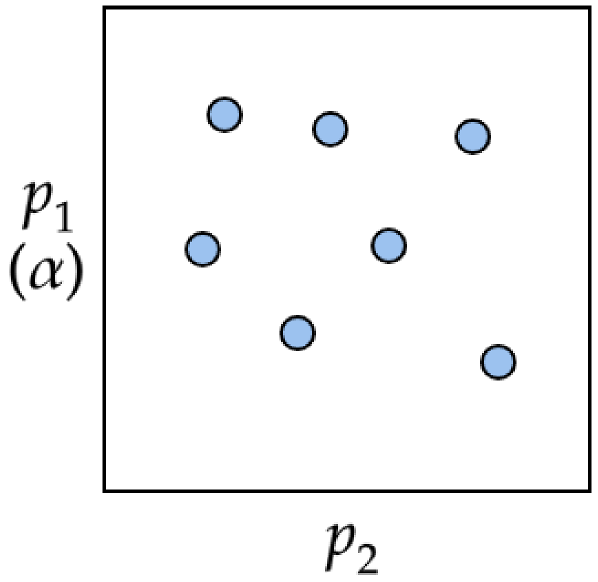
- Number of layers
- Number of nodes
- Learning rate decay
- Mini-batch size
- Momentum β
- Adams optimization $\beta_1,\ \beta_2$

Hyper-parameter space is very large, exhaustive grid search not practical.



- Only explores 3 values of p_1
- Grid search places equal importance on each parameter. That isn't true in practice.

Hyper-Parameter Tuning

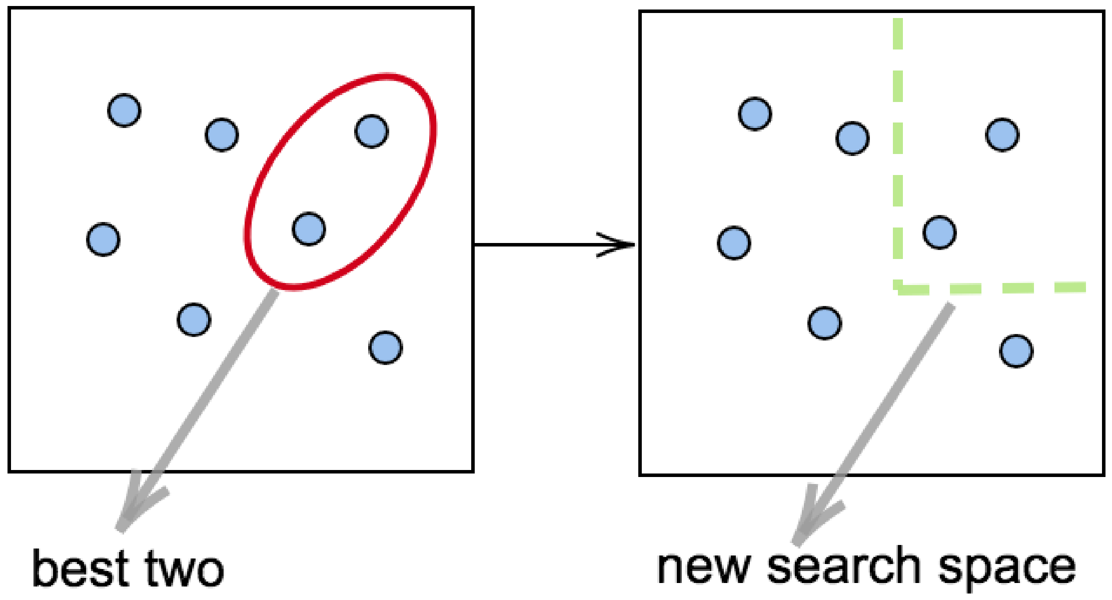


Random search is recommended.

- Now every trial is with a different value of α.

Hyper-Parameter Tuning

If subregions found promising, reduce search space.



Hyper-Parameter Tuning

Understand scale:

1.Number of layers

- Could just sample randomly from {2,3,4}

2.Number of nodes in a layer

- Again simple uniform sample from 50,...,100

3.Learning rate
Typical range is [.0001-1]. Uniform sampling would place 90% of sample in [.1 -1] and leave the smaller rates largely unexplored.

- Uniformly sample on a log scale. Choose $x \in [-4, 0]$ and $\alpha = 10^x$

4.Momentum β (exponential weighting parameter)

$\beta = .9 \sim 10 \quad \text{element average}$

$\beta = .999 \sim 1000 \quad \text{element average}$

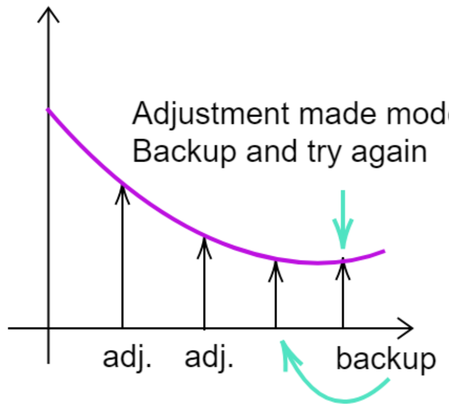
$$1 - \beta \in [.001 - .1]$$
$$\text{choose } x \in [-3, -1]$$
$$1 - \beta = 10^x$$
$$\beta = 1 - 10^x$$

Hyper-Parameter Tuning

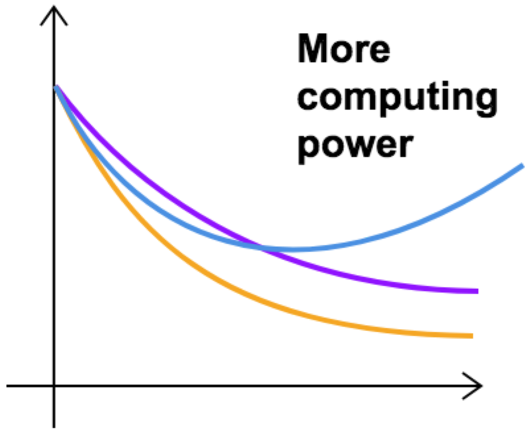
Hyper-parameters:
Application areas

- Vision (convolutional nets)
- NLP
- Speech

Approach #1:
Babysit a model *(Panda Strategy)*



Approach #2:
Train many models in parallel
(Caviar Strategy)



Automating trials using JSON configuration files

```
{
    "experiments":
    [{
        "name": "mnist",
        "train_path": "../../data/mnist",
        "test_path": "../../data/mnist",
        "sep": "\t",
        "configs": [
            {
                "name": "initial_experiment",
                "epochs": 10000,
                "batch_size": 10,
                "number_hidden": 5,
                "ks": [1],
                "momentums": [0.5, 0.6],
                "l_w": [0.05],
                "l_v": [0.05],
                "l_h": [0.05],
                "decay": 0.0002
            }
        ]
    }]
}
```

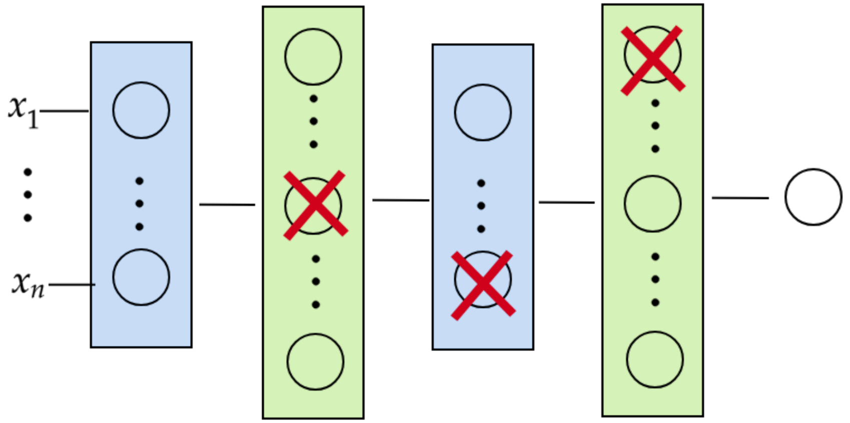
Dropout

Bagging-Bootstrap Aggregating[1]

- Sample data $\mathcal{D}$ with replacement.
- Train model on each sampled data set m-times.
- Average (regression) or vote (classification) to combine models.
- Gives improved performance if models have high variance.

[1] *Leo Breiman, 1994*

Dropout is a computationally efficient method to approximate bagging while optimizing a single set of parameters.



Randomly drop percentage of hidden nodes (make them output 0)

Dropout

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

- The potential to reduce overfitting is obvious because the effective network is much smaller (50%)

For each layer, define a dropout mask $\mathbf{d}^{[l]}$. Set element values to 0 with probability p.

$$\mathbf{d}^{[l]} = \begin{bmatrix} 0|1 \\ \vdots \\ 0|1 \end{bmatrix}$$

$$\frac{\sum d_i^{[l]}}{N_l} \approx 1 - p$$

$$\mathbf{a}^{[l]} = \mathbf{a}^{[l]} \odot \mathbf{d}^{[l]}$$

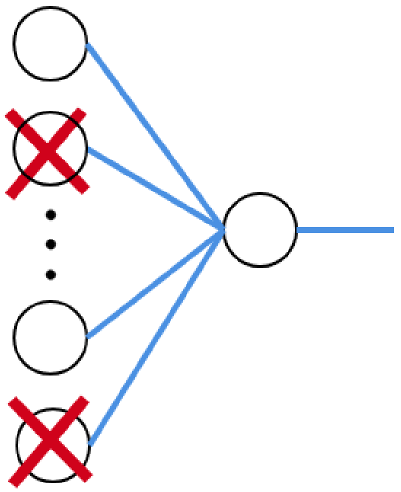
$$\mathbf{a}^{[l]} = \frac{\mathbf{a}^{[l]}}{p}$$

Dividing by p keeps expected value sent to next node the same as if no dropout

Dropout

- The mask vectors $\mathbf{d}^{[l]}$ are reset at each training step.
- At test time (model evaluation) no dropout is used.

Intuition



- Randomly removing nodes (weights) different at each training step, prevents reliance on a few nodes. The effect is to shrink weights, similar to L_2 regularization.
- Can be shown to be a type of adaptive (weight-by-weight) L_2 regularization.
- Dropout may be more sophisticated than simply adding noise.
- Getting ensemble behavior at no additional cost.

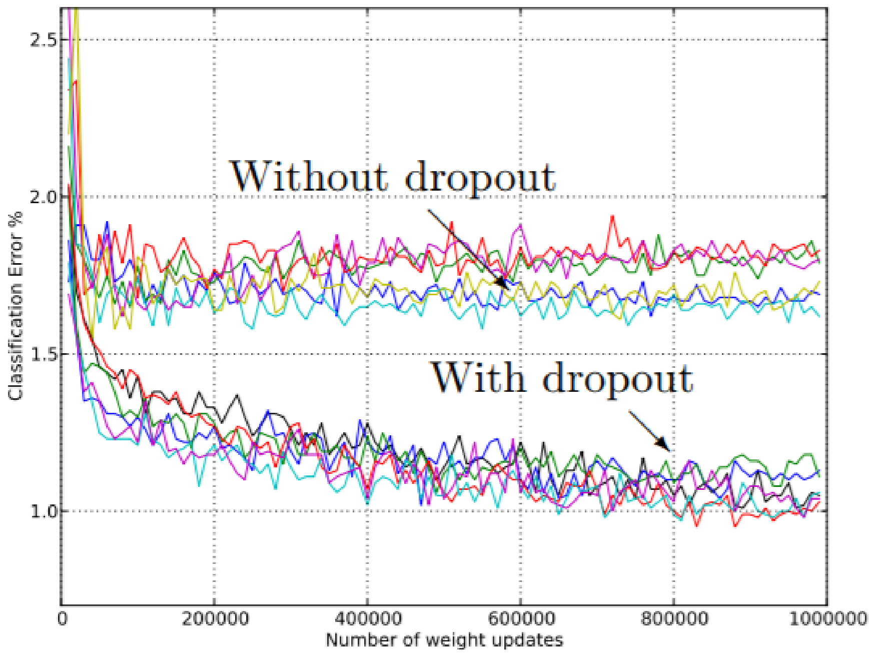
Dropout

Notes:

- Almost always used in computer vision.
- Requires lots of data.
- Only use if model is overfitting.
- Cost function is random, so more difficult to monitor.
- First run without dropout and verify that cost decreases - then enable dropout.

Dropout

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



Regularization

TBD - Regularization effect of adding noise

- Adding noise = Tykhonov
- Adding noise improves learning
- Adversarial noise

Netflix Prize

The Netflix Prize was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings without any other information about the users or films

Netflix provided a training data set of 100,480,507 ratings that 480,189 users gave to 17,770 movies. Each training rating is a quadruplet of the form (user, movie, date of grade, grade). Grades are from 1 to 5

Prizes were based on improvement over Netflix's own algorithm, called Cinematch, or the previous year's score if a team has made improvement beyond a certain threshold. A trivial algorithm that predicts for each movie in the quiz set its average grade from the training data produces an RMSE of 1.0540. Cinematch uses "straightforward statistical linear models with a lot of data conditioning"

The competition began on October 2, 2006. By October 8, a team called WXYZConsulting had already beaten Cinematch's results.

On June 26, 2009 the team "BellKor's Pragmatic Chaos", a merger of teams "Bellkor in BigChaos" and "Pragmatic Theory", achieved a 10.05% improvement over Cinematch

Netflix Prize

Jason Kempin/Getty Images Netflix prize winners, from left: Yehuda Koren, Martin Chabbert, Martin Piotte, Michael Jahrer, Andreas Toscher, Chris Volinsky and Robert Bell.

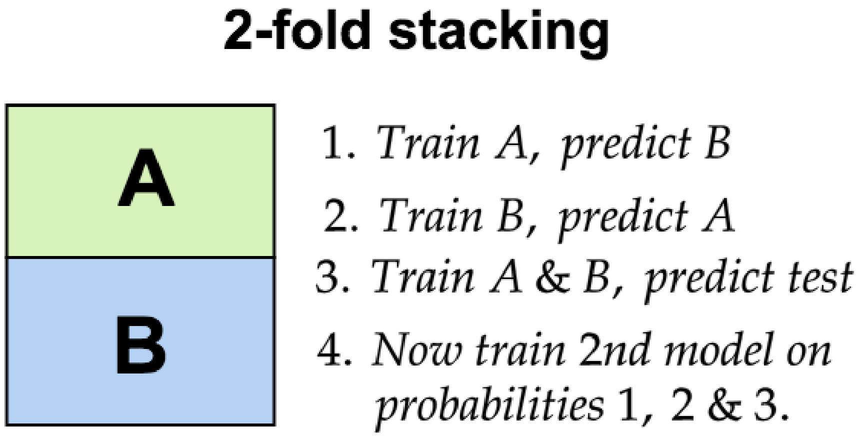
Netflix Prize

"This is how you win machine learning competitions: You take other people's work and ensemble them together."

Vitaly Kuznetsov, NIPS 2014

Kaggle Ensemble Guide

- Voting ensembles
- Averaging
- Stacking/Blending


2-fold stacking
A
B
1. Train A, predict B
2. Train B, predict A
3. Train A & B, predict test
4. Now train 2nd model on probabilities 1, 2 & 3.

Ensemble Methods

Ensemble methods train several different (type of) models. In testing, ensemble methods average the output of these models.

- The basic intuition between ensemble methods is that if models are independent, they will usually not all make the same errors on the test set.

- With K independent models, the average model error will decrease by a factor of $1/k$. Denoting ϵ_i to be the error of model i on an example, and assuming $E\epsilon_i = 0$ as well as that the statistics of this error is the same across all models,

$$\mathbf{E}\left[\left(\frac{1}{k}\sum_{i=1}^{k}\epsilon_i\right)^2\right] = \frac{1}{k^2}\sum_{i=1}^{k}\mathbf{E}{\epsilon_i}^2 = \frac{1}{k}\mathbf{E}{\epsilon_i}^2$$

- If the models are not independent the ensemble expected error is:

$$\frac{1}{k}\mathbf{E}{\epsilon_i}^2 + \frac{k-1}{k}\mathbf{E}\left[\epsilon_i\epsilon_j\right]$$

which is equal to $E\epsilon_i^2$ only when the models are perfectly uncorrelated.

Ensemble Methods

Averaging

Model 1: quality q_1, estimates $\hat{y}_1$

Model 2: quality q_2, estimates $\hat{y}_2$

$$\hat{y} = \frac{q_1\hat{y}_1 + q_2\hat{y}_2}{q_1 + q_2}$$

Ensemble Methods

Stacking

For models $\mu_1, \ldots, \mu_n$

Divide training data into K folds.

Train K "leave-one-out" models.

Train a subsequent "stacked" model on the output predictions of the "leave-one-out" models.

$$\begin{bmatrix} \hat{y}_{F_1}^{\mu_1} & \cdots & \hat{y}_{F_1}^{\mu_n} \\ \vdots & & \vdots \\ \hat{y}_{F_k}^{\mu_1} & \cdots & \hat{y}_{F_k}^{\mu_n} \end{bmatrix}$$