Lecture 2* Optimization and Regularization in Deep Learning

课程：机器学习与深度学习

Recap of the Last Lecture

- Model Architectures
  - Artificial neurons
  - Activation function and saturation
  - Feedforward neural nets
- How to train a neural net
  - Loss Function Design
  - Optimization
    - Gradient Descent and Stochastic Gradient Descent
    - Back-propagation

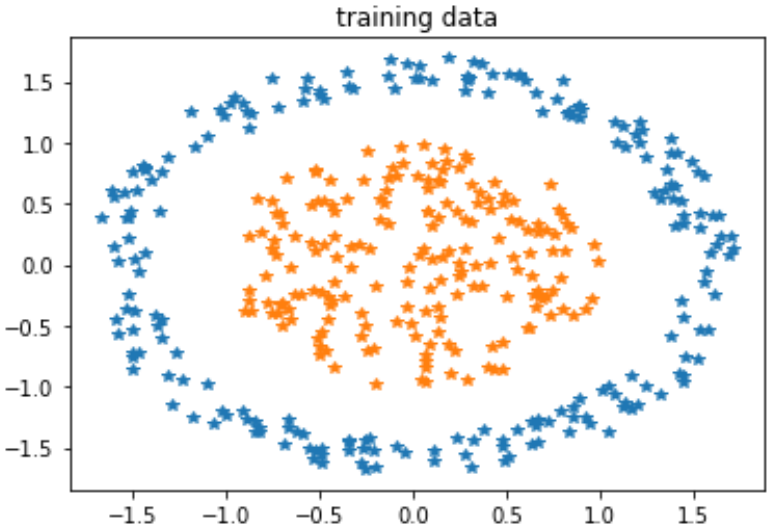
Two Important Questions in DL

- How to train large scale neural nets? (Efficiency)
  - Algorithm: SGD, momentum, Adagrad, etc.
  - Architecture: multicore CPU, GPU, Spark, TensorFlow, H2O, etc.
- How to improve generalization and prevent overfitting? (Effectiveness)
  - Norm regularizer/constraint
  - Ensemble
  - Dropout
  - …

Optimization in Neural Nets

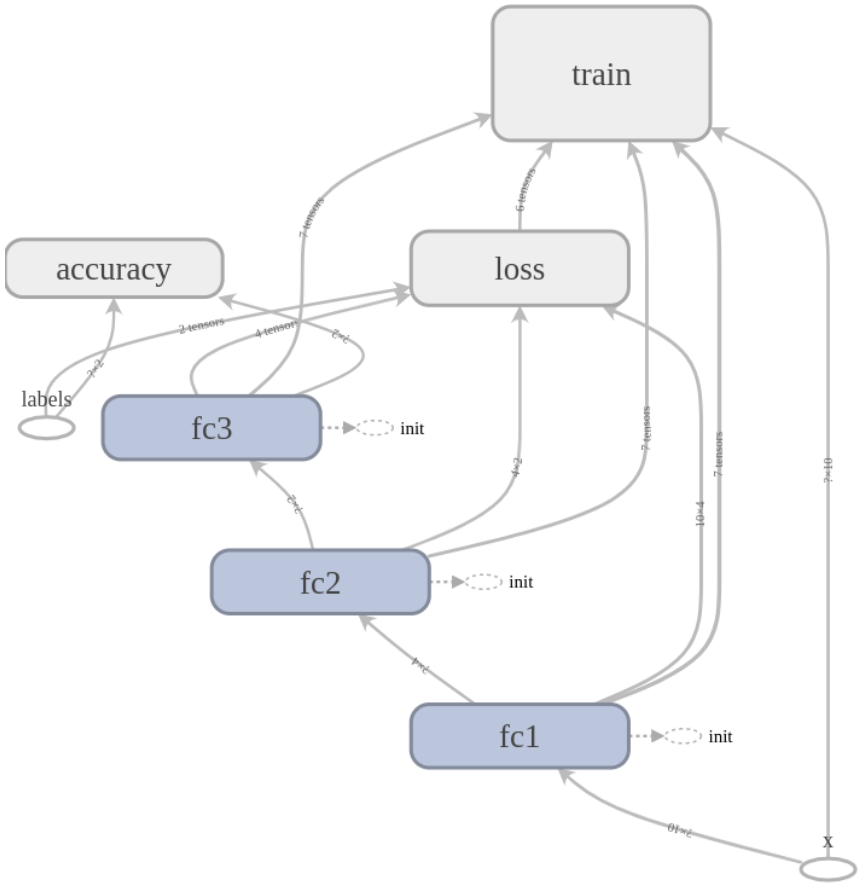
Start with an easy example

- Given $(x_1, x_2, \ldots, x_d)$ and their true label y, where y is either 0 or 1, build a neural network to predict true label.



Start with an easy example

- Three layers

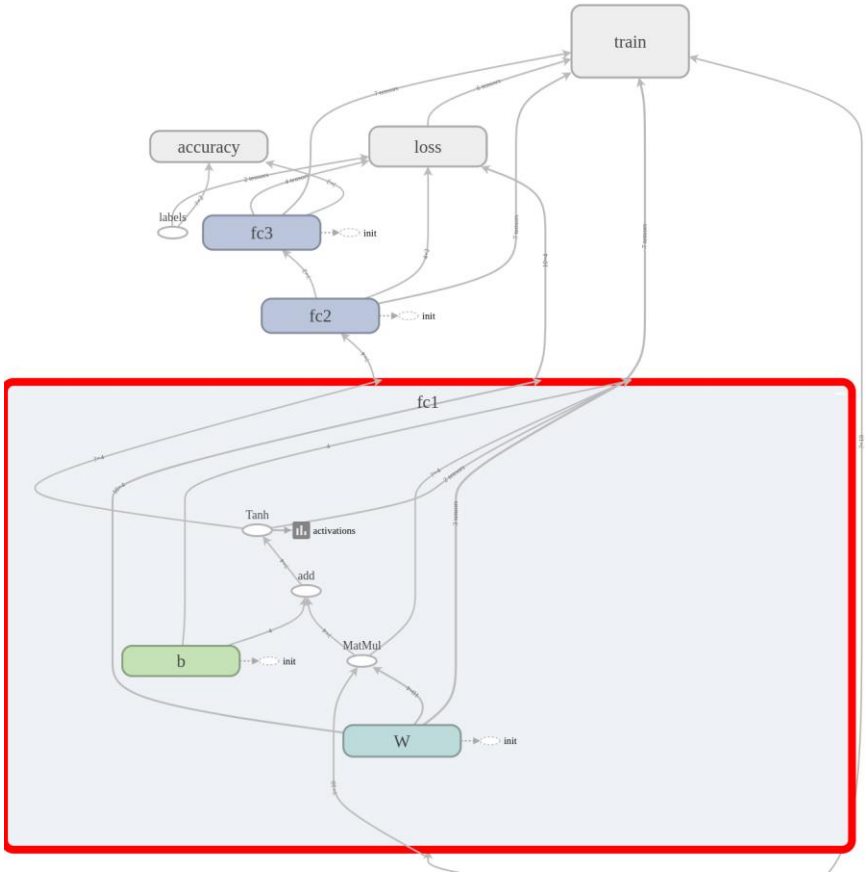


Start with an easy example

- Three layers
- Each layer:

$$y = \tanh(Wx + b)$$

```
def fc_layer(inputs, input_sz, output_sz):
  W = tf.get_variable(
    'W', [input_sz, output_sz],
    initializer=tf.zeros_initializer()
    )
  b = tf.get_variable(
    'b', [output_sz],
    initializer=tf.zeros_initializer()
    )
  outputs = tf.tanh(tf.matmul(inputs, W) + b)
  return outputs
```

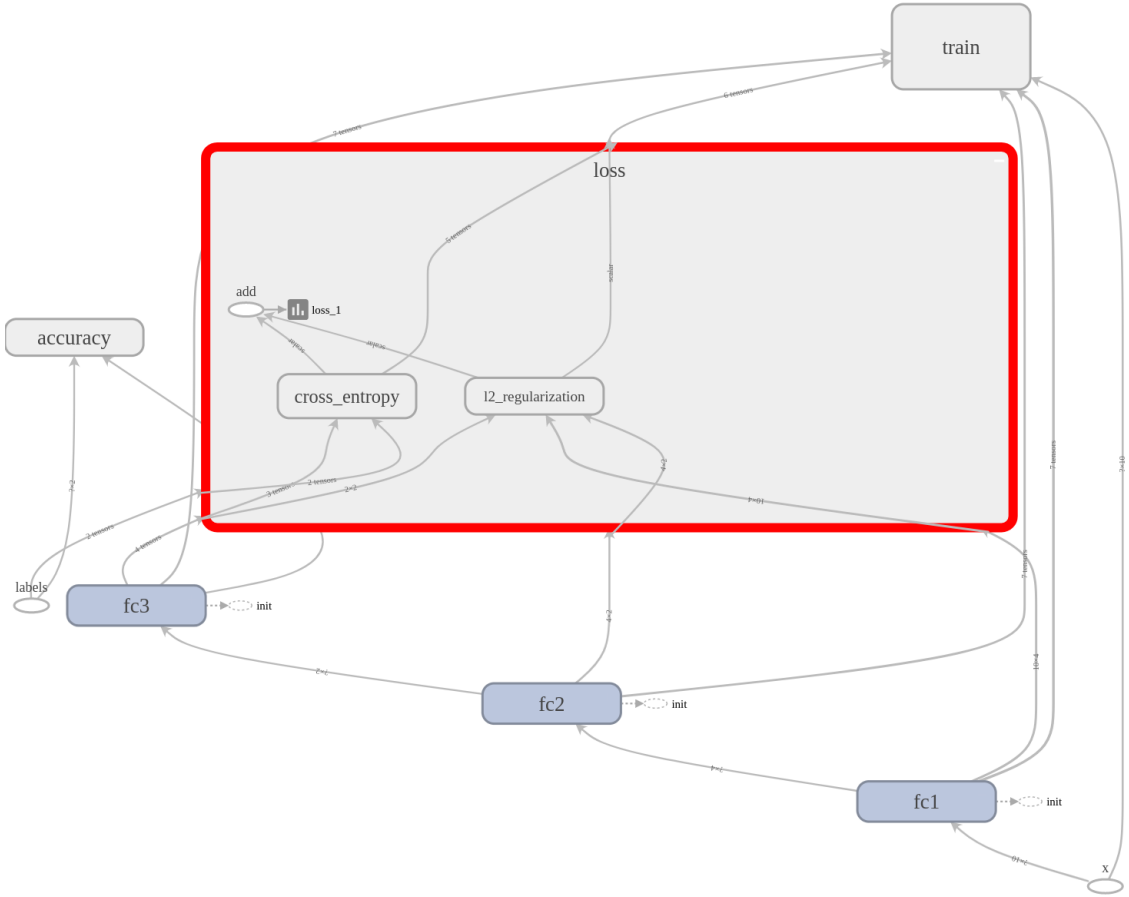


Start with an easy example

- Three layers
- Each layer:

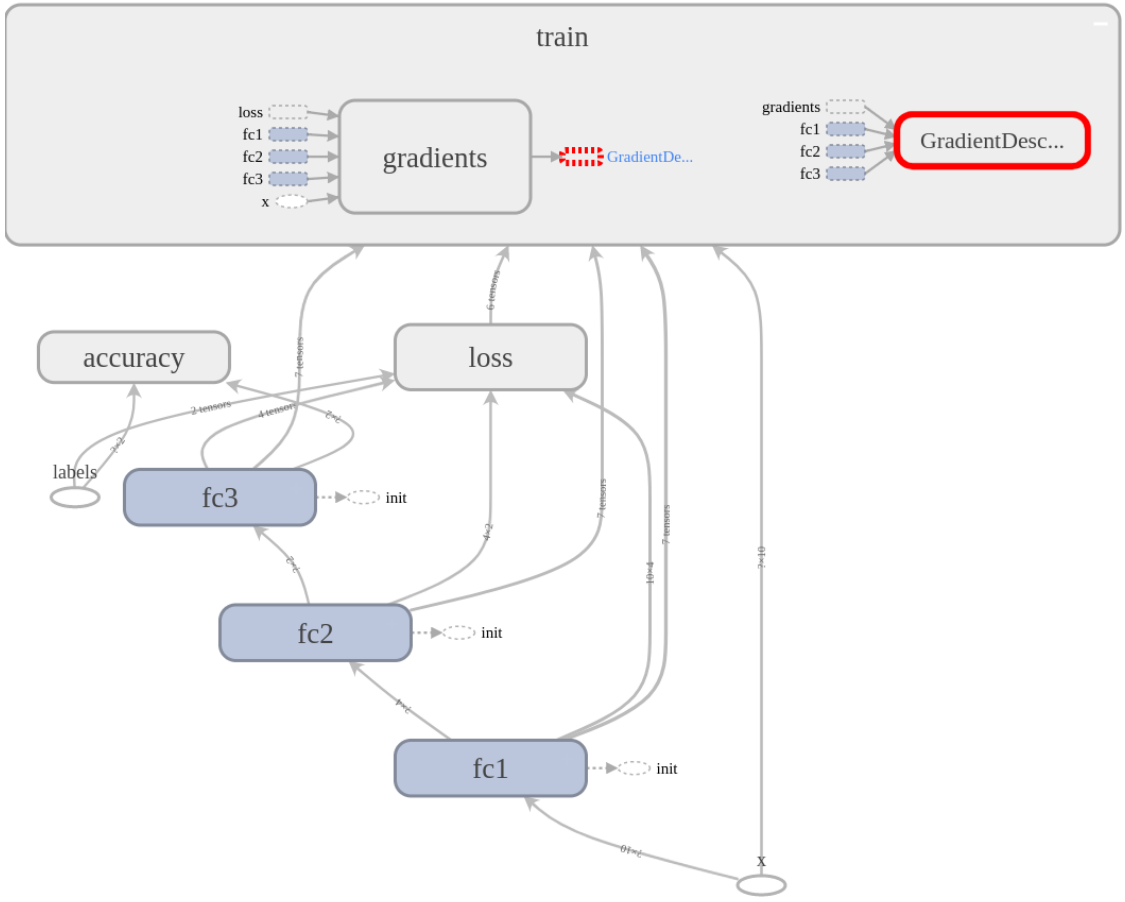
$y = \tanh(Wx + b)$

- Loss
  - softmax loss
  - L2 regularization



Start with an easy example

- Three layers
- Each layer:
  $y = \tanh(Wx + b)$
- Loss
  softmax loss
  L2 regularization
- Optimizer
  Gradient descent



Gradient Descent

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} l\left(f\left(x^{(i)}, \theta\right), y^{(i)}\right) + \lambda \Omega(\theta)$$

- Gradient descent

for $t = 1,2,3, \ldots$

$$\mathbf{g}^{(t)} = \frac{1}{n} \sum_{i}^{n} \nabla l\left(f\left(\mathrm{x}^{(i)}, \boldsymbol{\theta}^{(t)}\right), y^{(i)}\right) + \lambda \nabla \Omega\left(\boldsymbol{\theta}^{(t)}\right)$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \mathbf{g}^{(t)}$$

- Stochastic gradient descent

for $t = 1,2,3, \ldots$

sample $i \in \{1,2, \ldots, n\}$

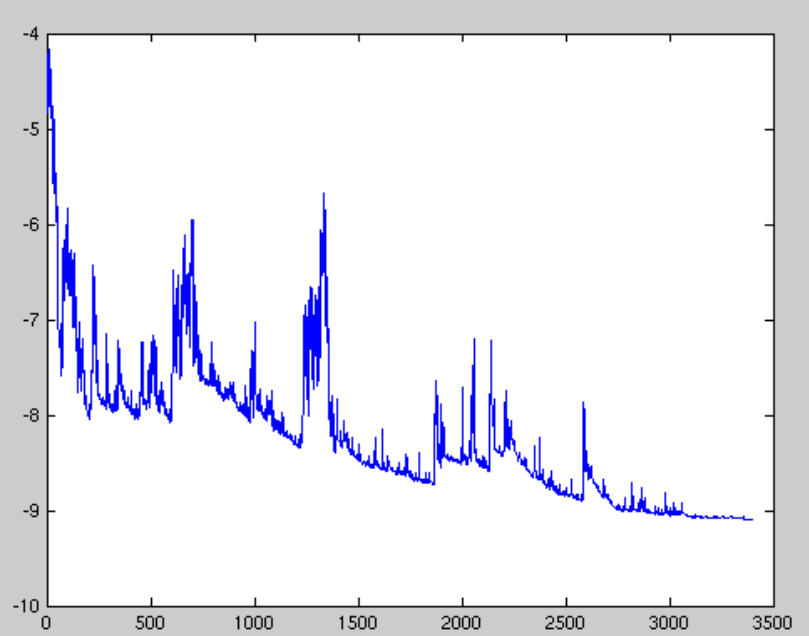
$$\mathbf{s} = \nabla l\left(f\left(\mathrm{x}^{(i)}, \boldsymbol{\theta}^{(t)}\right), y^{(i)}\right) + \lambda \nabla \Omega\left(\boldsymbol{\theta}^{(t)}\right)$$

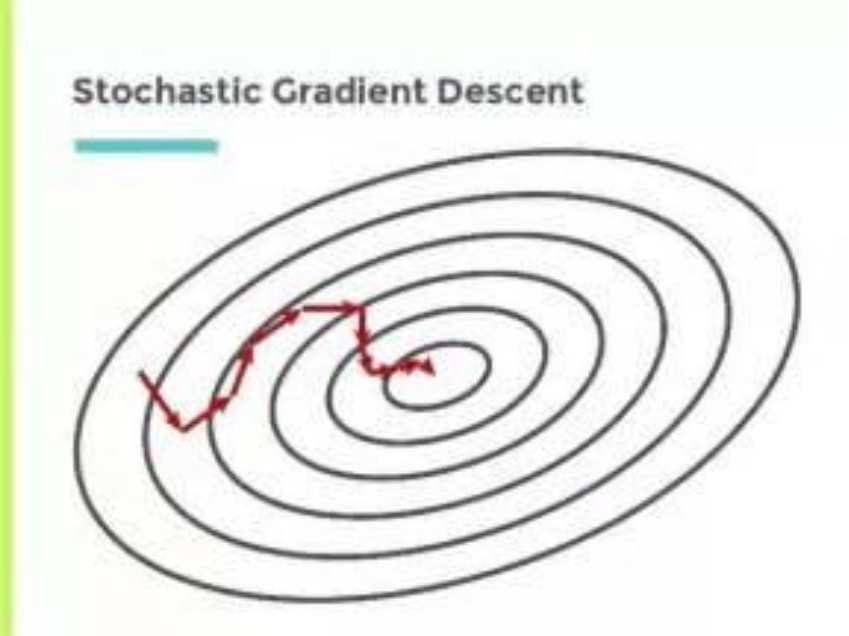
$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \cdot \mathbf{s}$$

SGD vs GD

- SGD: use a single data in each iteration
- GD: use full dataset in each iteration
- If the data is highly redundant (small variance), gradient on the first half of dataset is almost identical to the gradient on the second half
- We prefer SGD, but often employ mini-batch SGD in practice
  - can take advantages of matrix matrix multiplication in GPU

SGD


-4
-5
-6
-7
-8
-9
-10
0
500
1000
1500
2000
2500
3000
3500

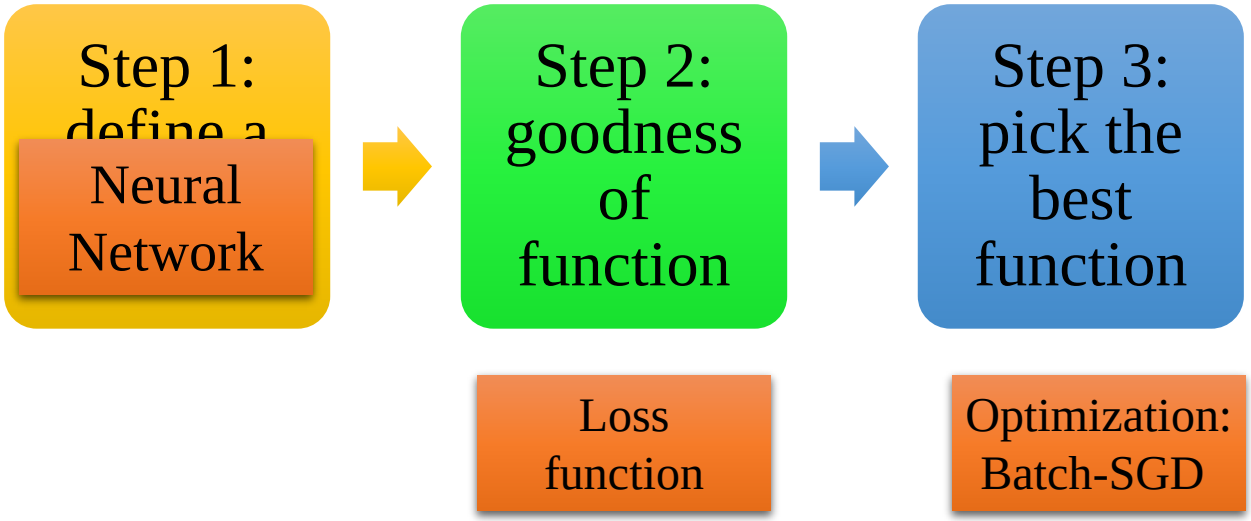

Stochastic Gradient Descent

Mini-batch

- Update based on a mini-batch of training data
  - can produce a more accurate estimate of the gradient
  - can leverage matrix/matrix operations, which are more efficient in GPUs
- Sample a mini-batch $I_t \subseteq \{1,2,\dots,n\}$ and compute

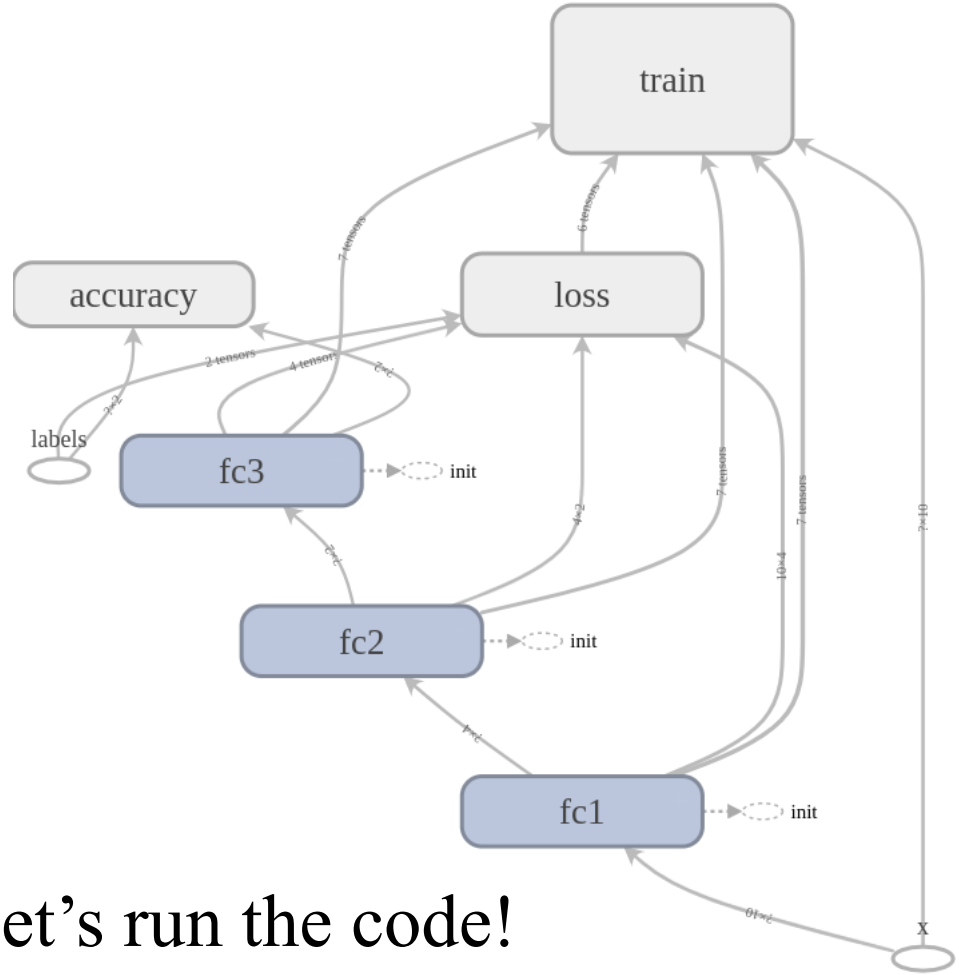
$$g_t = \frac{1}{|\mathrm{I_t}|}\sum_{i \in I_t} \nabla l\left(f\left(\mathrm{x}^{(i)}, \boldsymbol{\theta}^{(t)}\right), y^{(i)}\right) + \lambda \nabla \Omega\left(\boldsymbol{\theta}^{(t)}\right)$$

Three Steps for Deep Learning



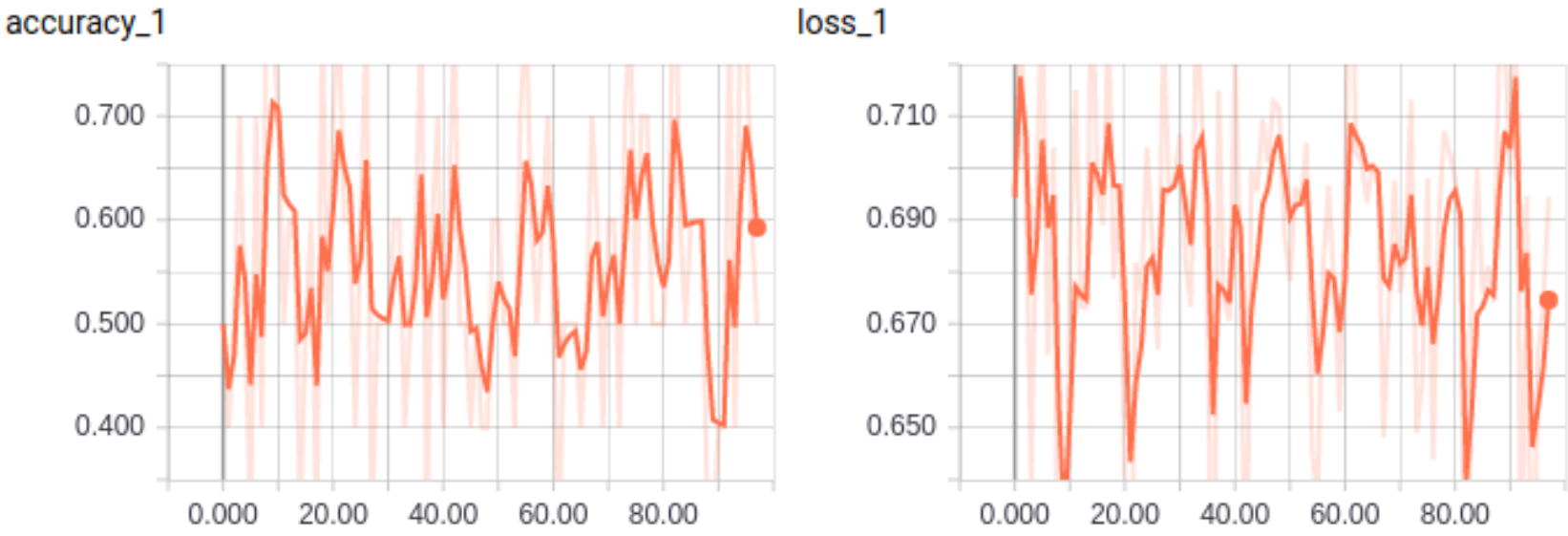
Start with an easy example

- Three layers
- Each layer:
  $y = \tanh(Wx + b)$
- Loss
  softmax loss
  L2 regularization
- Optimizer
  Gradient descent

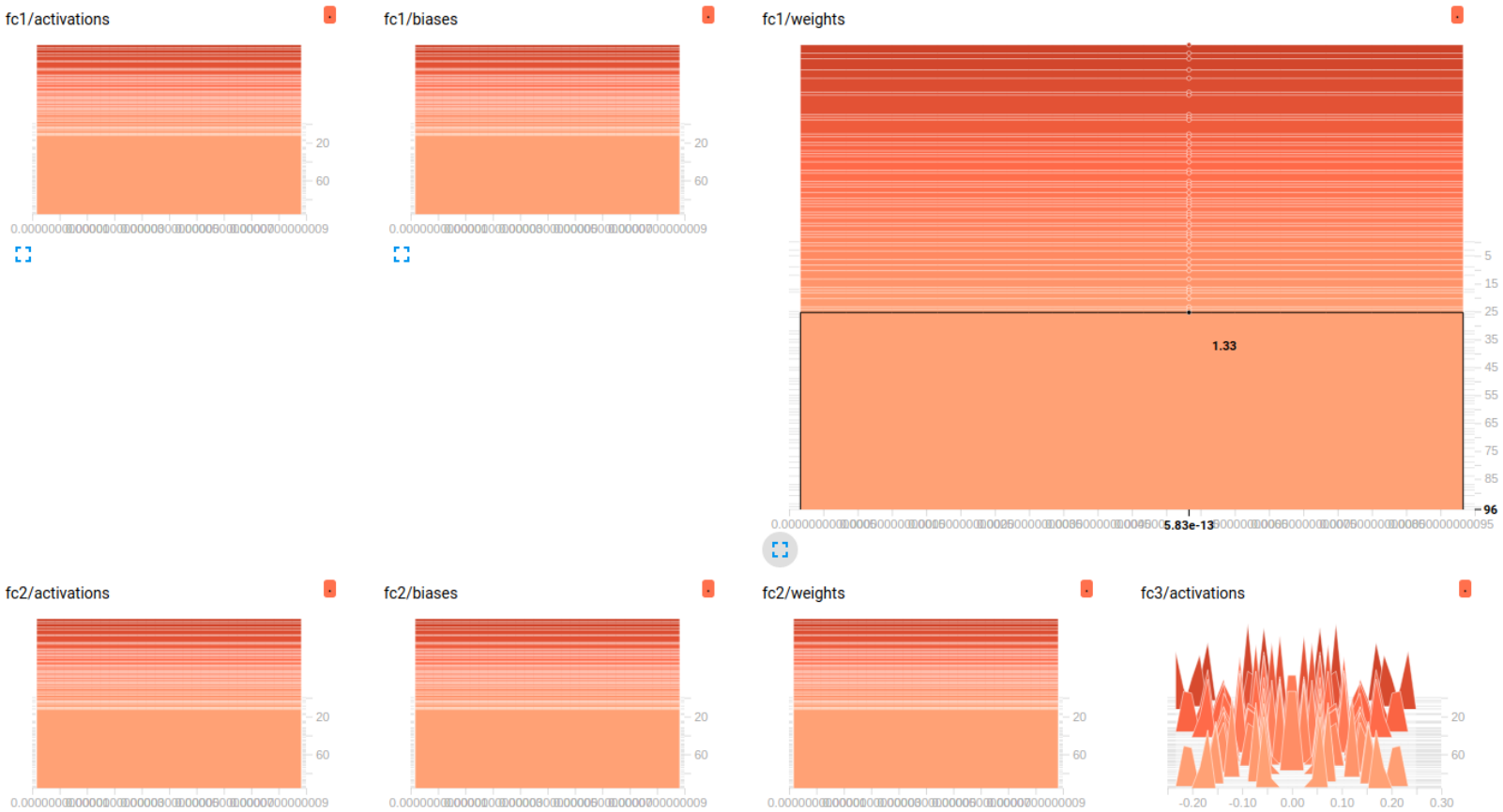


Everything seems fine, let's run the code!

Oops!


accuracy_1
0.700
0.600
0.500
0.400
0.000
20.00
40.00
60.00
80.00
loss_1
0.710
0.690
0.670
0.650
0.000
20.00
40.00
60.00
80.00

What goes wrong?



Some strategies in optimization

- **Weight initialization**
- Preprocessing
- Other tips

Initialization

```python
def fc_layer(inputs, input_sz, output_sz):
  W = tf.get_variable(
    'W', [input_sz, output_sz],
    initializer=tf.zeros_initializer()
    )
  b = tf.get_variable(
    'b', [output_sz],
    initializer=tf.zeros_initializer()
    )
  outputs = tf.tanh(tf.matmul(inputs, W) + b)
  return outputs
```

See layer 2 as an example:

Denote layer 1's output as h, then

$$Loss = f(Wh + b) + \lambda||W||2$$

$$Gradient = f'(Wh + b)h^T + 2\,\lambda W$$

Since $h, W = 0$, gradient will always be 0, i.e. update never happens!

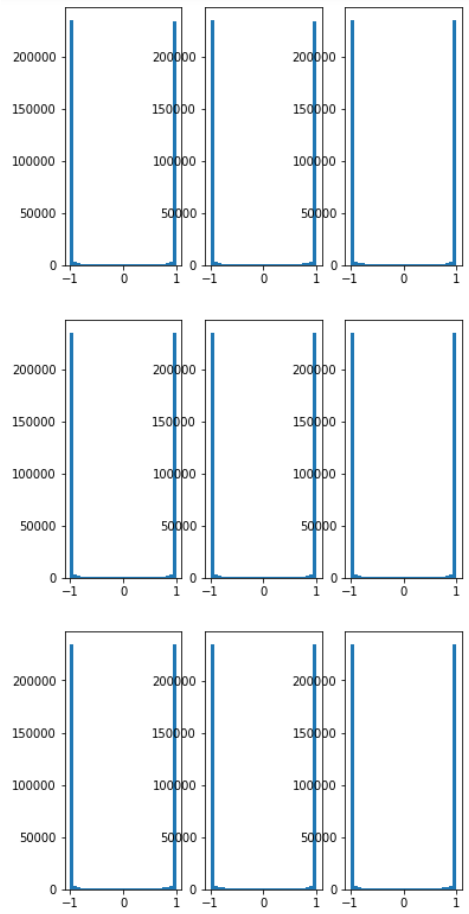
- Bias can generally be initialized to zero.
- What about weight matrix?

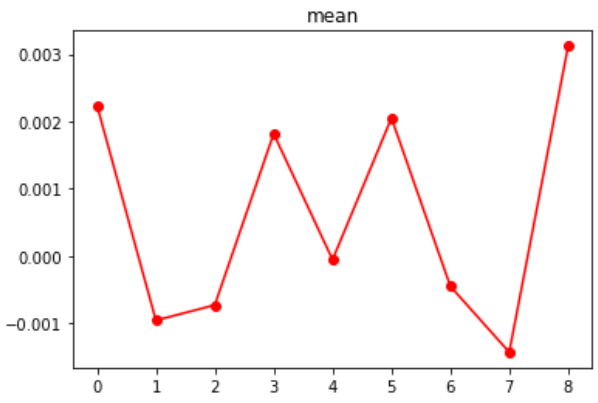
If we use all zero initialization, all the fprop and bprop compute the same value.

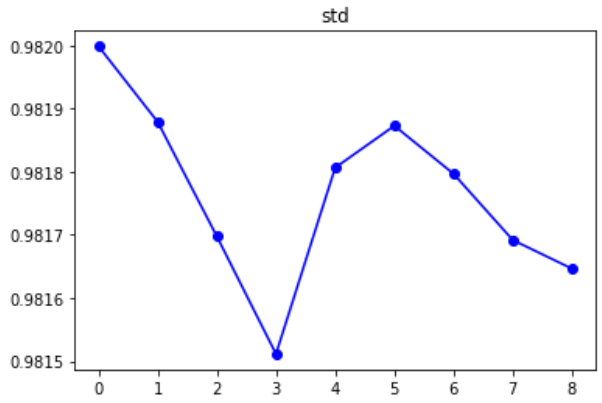
Weight matrix initialization

- What about initializing weights with Gaussian Noise?

```
inputs = np.random.randn(1000, 500)
hidden_layer_sizes = [500] * 9

Hs = []
for i in range(len(hidden_layer_sizes)):
    X = inputs if i == 0 else Hs[i - 1]
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out)
    Hs.append(np.tanh(X.dot(W)))
```

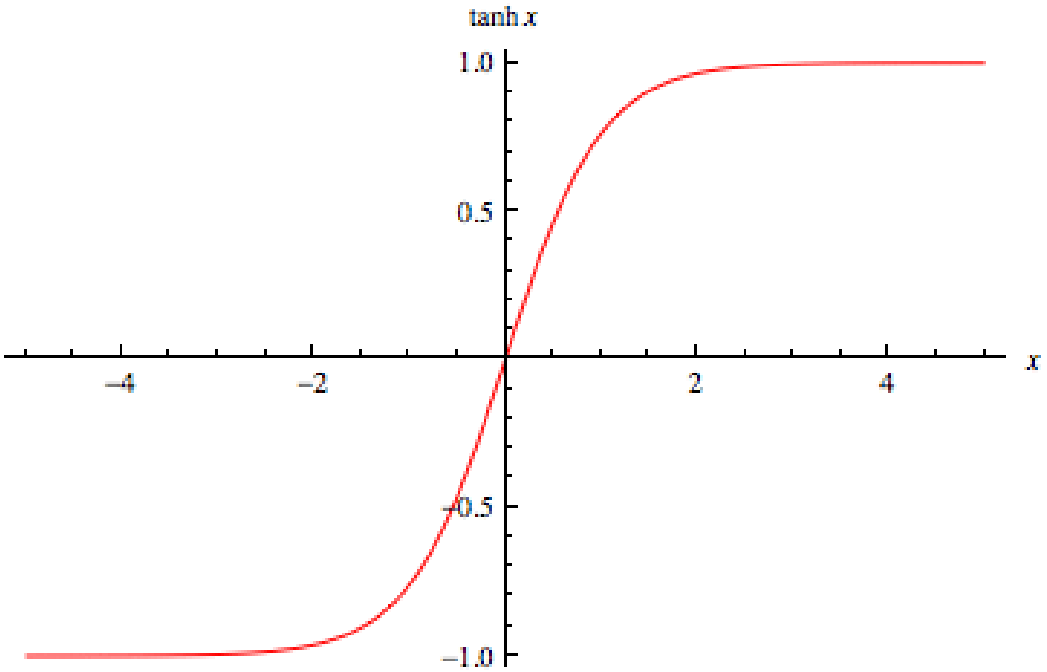


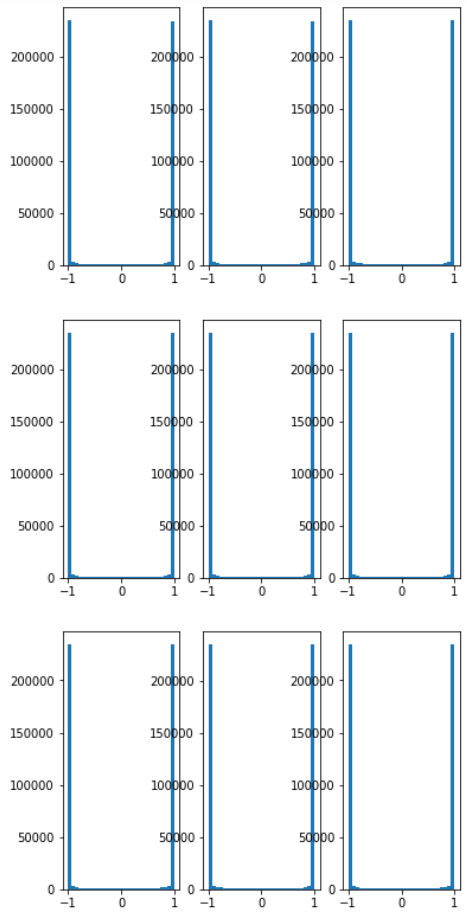




Weight matrix initialization

- What about initializing weights with Gaussian Noise?



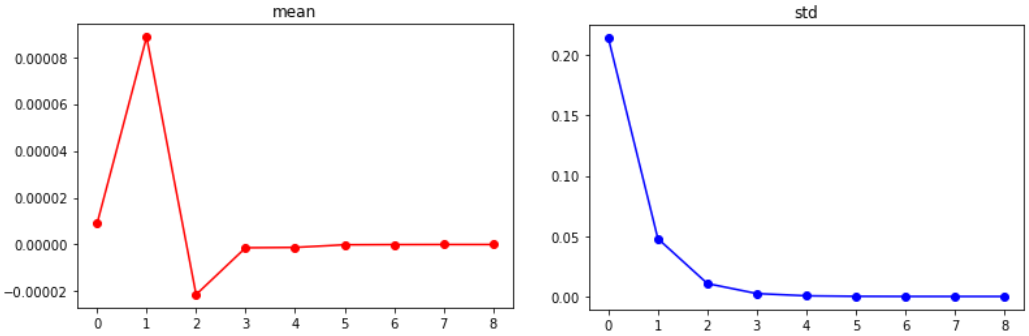


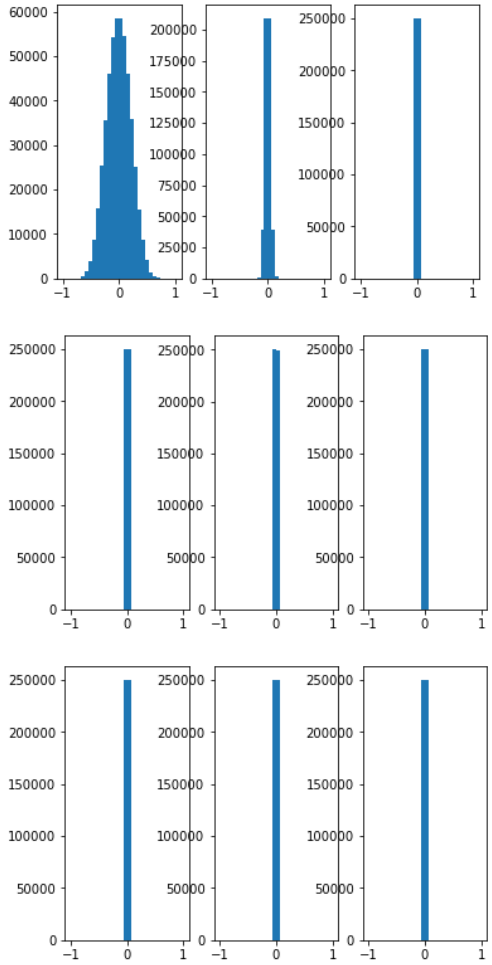
Gradient Saturation!

Weight matrix initialization

- What about initializing with Gaussian Noise * small value?

```
inputs = np.random.randn(1000, 500)
hidden_layer_sizes = [500] * 9

Hs = []
for i in range(len(hidden_layer_sizes)):
    X = inputs if i == 0 else Hs[i - 1]
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(
      fan_in, fan_out) * .01
    Hs.append(np.tanh(X.dot(W)))
```

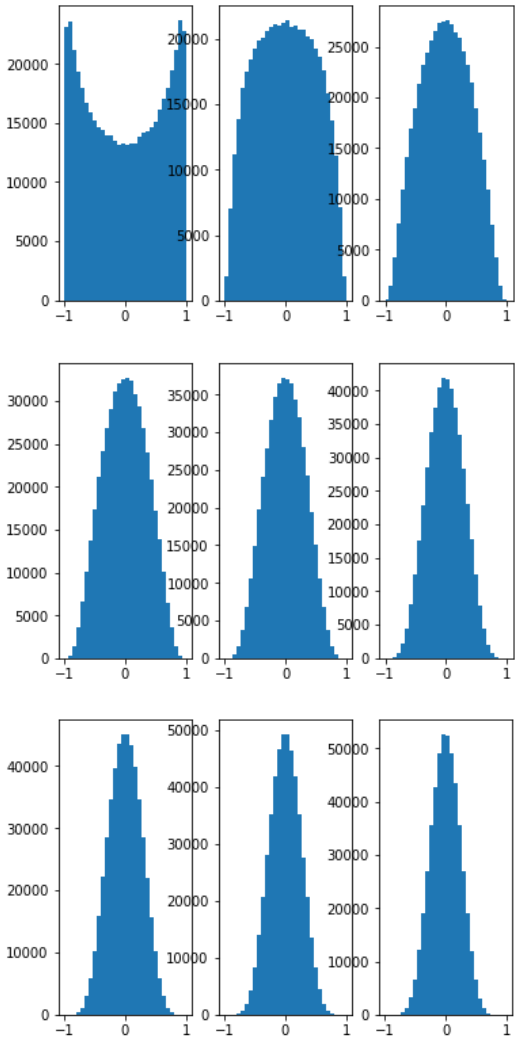




Like zero initialization,
gradients are killed in deep layer!

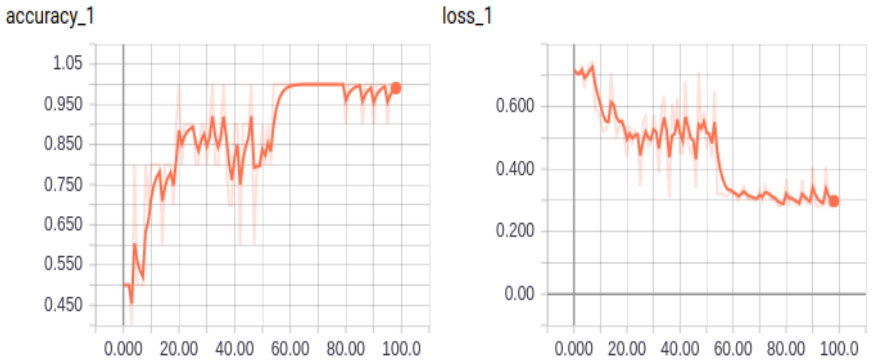
Weight matrix initialization

- The idea
  - Initialize weights with small random values to break symmetry between hidden units of the same layer.
    - e.g. Gaussian / sqrt(fan_in)



Weight matrix initialization

- The idea
  - Initialize weights with small random values to break symmetry between hidden units of the same layer.
    - e.g. Gaussian / sqrt(fan_in)




```
100%|██████████| 100/100 [00:08<00:00, 12.25it/s]
Test accuracy is 0.970000.
```

Weight Matrix Initialization

- Other initialization
  - Initialize $W_{ij}^{(k)}$ from $U[-b, b]$, where $b = \frac{\sqrt{6}}{\sqrt{n_j + n_{j-1}}}$
  - Xavier initialization [Glorot et al., 2010], $b = \frac{1}{\sqrt{n_{j-1}}}$
  - Not so effective in ReLU, recommend $b = \frac{1}{\sqrt{n_{j-1}/2}}$ [He et al., 2015]

Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks[C]//Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010: 249-256.

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." *Proceedings of the IEEE international conference on computer vision*. 2015.

Mishkin, D., & Matas, J. (2015). All you need is a good init. arXiv preprint arXiv:1511.06422.

Zhang, Hongyi, Yann N. Dauphin, and Tengyu Ma. "Fixup Initialization: Residual Learning Without Normalization." arXiv preprint arXiv:1901.09321 (2019).

Batch Normalization

- When discussing about initialization, we want each layer's output to look like Gaussian, so that gradient won't be killed or get saturated
- What if we force it to act like that by hand?

Ioffe S, Szegedy C. Batch normalization: Accelerating deep network training by reducing internal covariate shift[J]. arXiv preprint arXiv:1502.03167, 2015.
Ioffe S. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models[C]//Advances in Neural Information Processing Systems. 2017: 1942-1950.

Batch Normalization

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

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

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

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

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

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

Batch Normalization: Condition Number

- Think about linear model

$$\min_{w} \|y - Xw\|^2$$

  Iteration complexity of gradient descent:

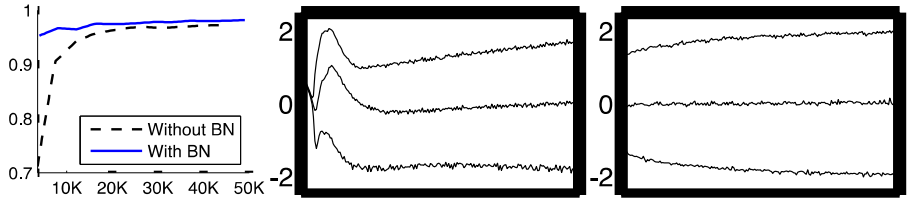
$$\kappa \log(\frac{1}{\epsilon})$$

  where $\kappa = \frac{\lambda_{max}(X^T X)}{\lambda_{min}(X^T X)}$

- In deep nets, X can be the output of a hidden layer (H) and κ covers a wide range
- In contrast to convex opt, it is very difficult to choose learning rates when the context is changing

Batch Normalization

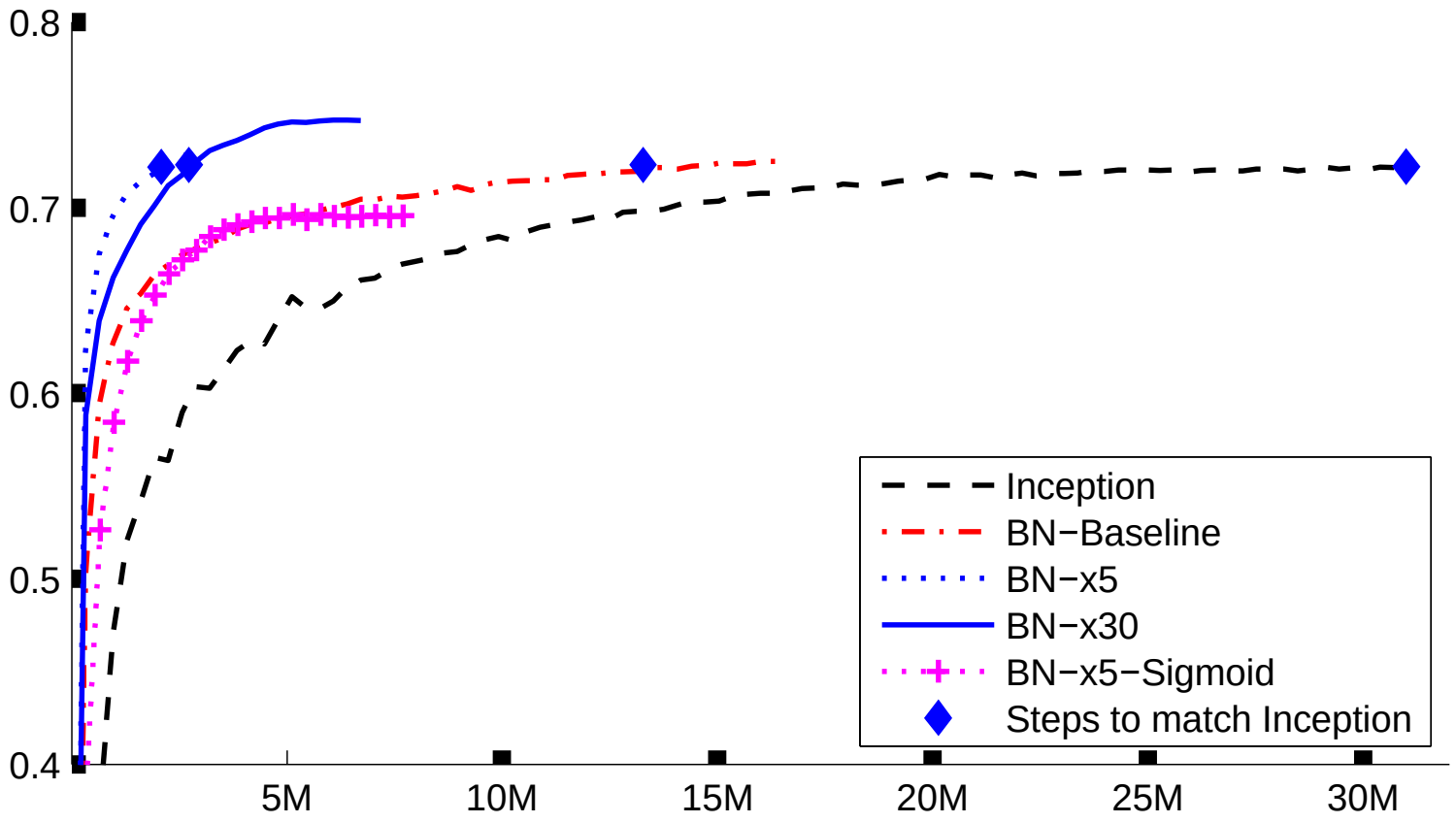
- BN provides an efficient way to re-parameterize deep nets
  - Improve the gradient flow through the network
  - More stability
  - Allows larger learning rate
  - Smaller L^2 weight
  - Higher accuracy
  - Reduce the strong dependence on initialization
  - Can be seen as a way to do regularization, which reduces the need for dropout



(a) (b) Without BN (c) With BN

a) Accuracy on MNIST data
b), c) The evolution of input distributions to a typical sigmoid

Batch Normalization



Other Normalizations

- Wu, Y., & He, K. (2018). **Group normalization**. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 3-19).
- Lei Ba, J., Kiros, J. R., & Hinton, G. E. (2016). **Layer normalization**. *arXiv preprint arXiv:1607.06450*.
- Ioffe, Sergey. "**Batch renormalization**: Towards reducing minibatch dependence in batch-normalized models." *Advances in neural information processing systems*. 2017.

Some strategies in optimization

- Weight initialization
- **Preprocessing**
- Other tips

Preprocessing

1. Subtract each dimension with its training mean (e.g. images)
2. Process the covariance matrix
   - ① Normalize features: subtract mean, scale to have unit variance
   - ② Whitening features: covariance matrix to be identity matrix
   - ③ Decorrelate features: covariance matrix to be diagonal matrix
3. PCA

Some strategies in optimization

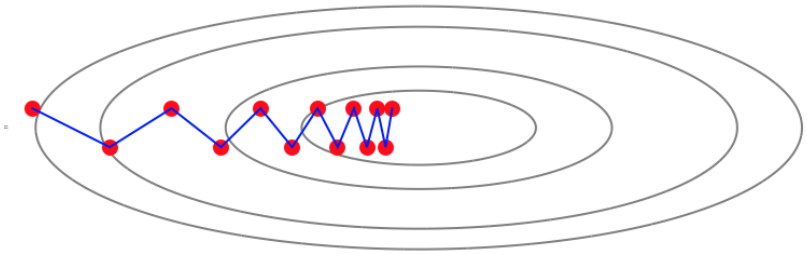
- Weight initialization
- Preprocessing
- **Other tips**

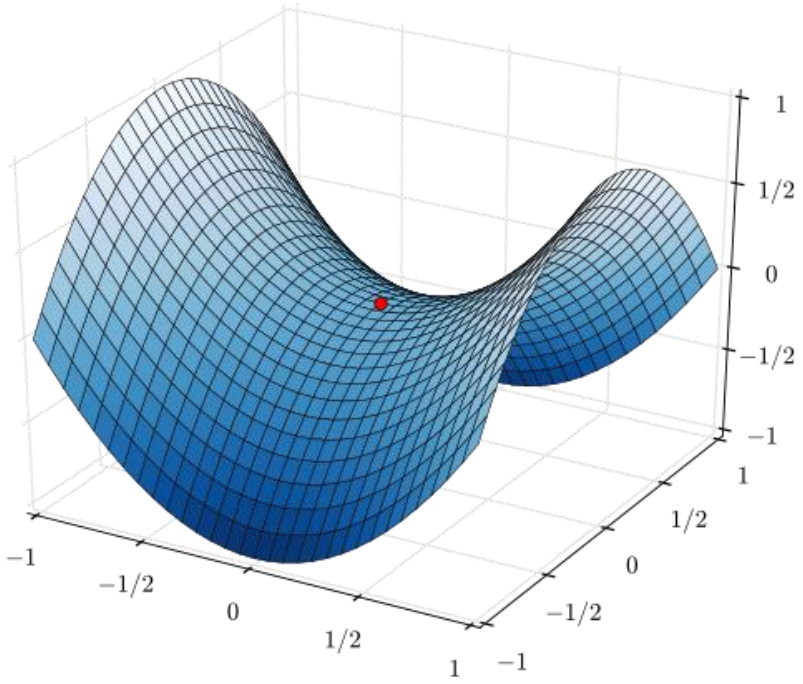
Other tips

- Randomly initialize hyperparameters
- Tune up regularization strength if validation accuracy is much lower than training accuracy

More about SGD

- Drawbacks
  - Large condition number
  - Local minima / saddle point
  - Noise due to stochastic





Momentum

- Gradient descent uses the gradient to change the position,

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta)$$

- Instead of using the gradient to change the *position* of θ, use it to change the "*velocity*".

$$v_{t+1} = \mu v_t - \eta \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

  - accelerate progress along dimensions in which gradient consistently point in the same direction
  - slow progress along dimensions where the sign of the gradient continues to change
  - μ ("momentum") dampens the velocity and reduces the kinetic energy of the system

Intuition

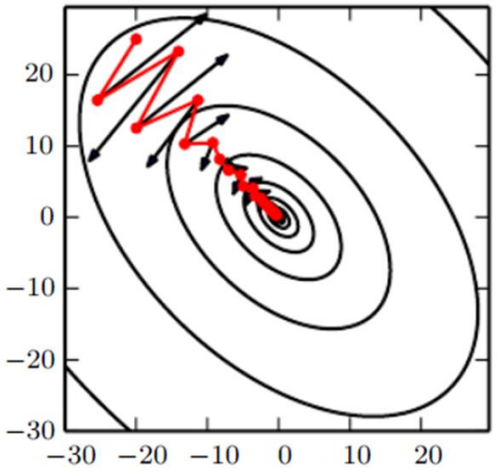
- Integrate velocity

$$v_{t+1} = \mu v_t - \eta \nabla J(\theta_t) \Delta t$$

- Integrate position

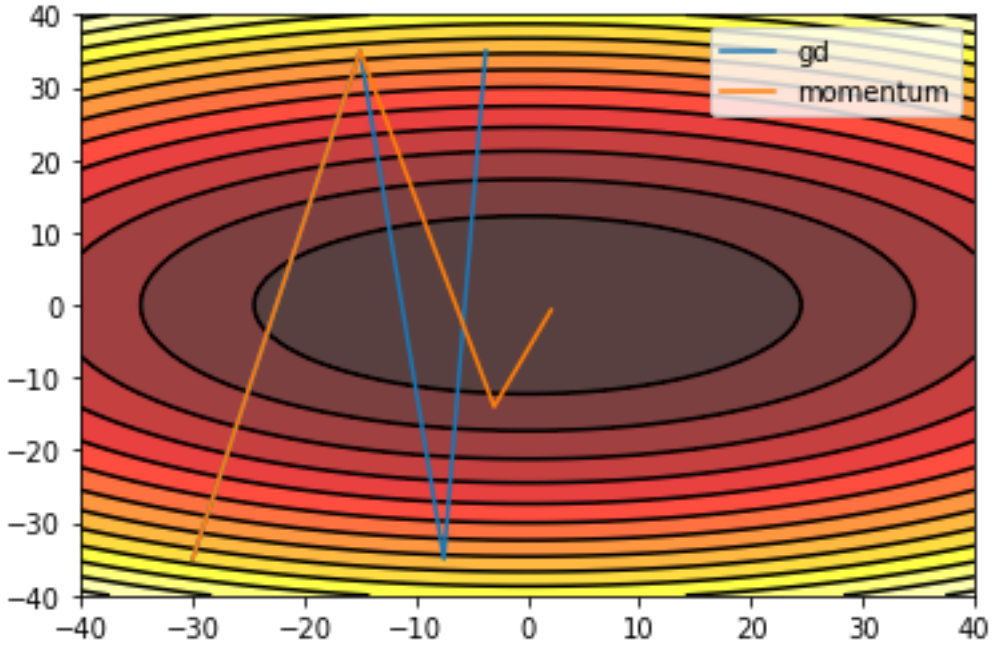
$$\theta_{t+1} = \theta_t + v_{t+1} \Delta t$$

- μ ("momentum") dampens the velocity and reduces the kinetic energy of the system



Advantage

- Escape Local minima / Saddle point by remained kinetic
- Deal with large condition number: damp

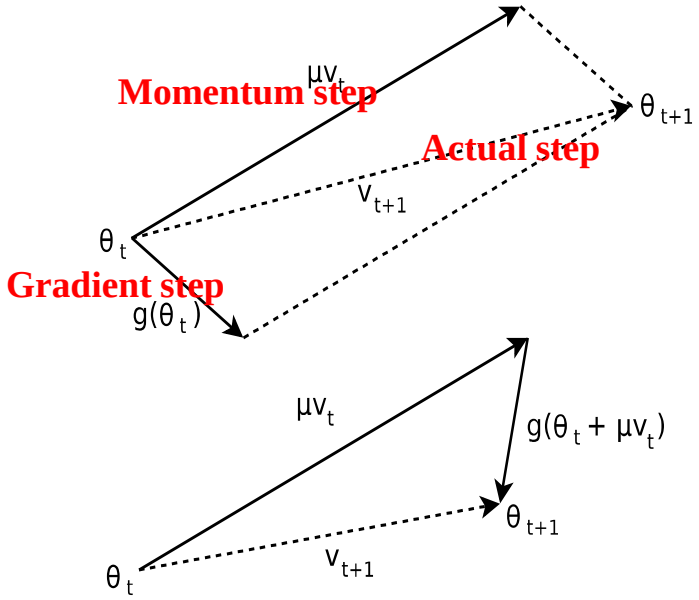


Nesterov's Method

- Nesterov's accelerated gradient method

$$v_{t+1} = \mu v_t - \eta \nabla J(\theta_t + \mu v_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

- NAG allows quicker and more responsive feedback
- In the convex setting, NAG achieves $O(\frac{L}{T^2} + \frac{\sigma}{\sqrt{T}})$ while GD achieves $O(\frac{L}{T} + \frac{\sigma}{\sqrt{T}})$.



Top: Classical Momentum
Bottom: NAG
to optimize $\min_{\theta} g(\theta)$

Adagrad

- Learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

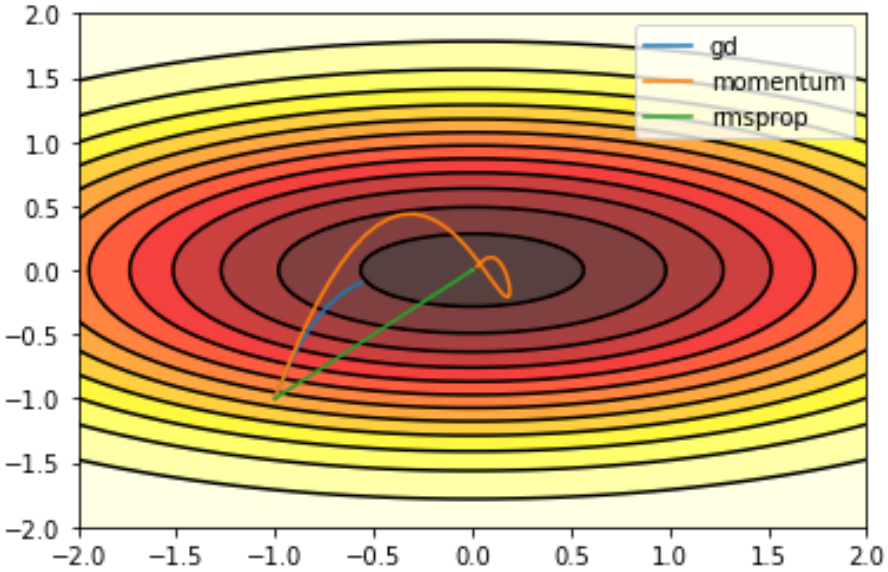
- G_t contains the sum of square of the past gradient $G_t = G_{t-1} + g_t^2$
- ϵ is a small constant to prevent division by zero
- Advantage
  - Solve poor condition number problem
- Disadvantage
  - Step size goes smaller, it will stop at saddle point

Adadelta

- Accumulate sum of square of gradient over a window
$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2,$$
ρ is a constant so that the weight of past gradient is diminishing
- $\text{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon}$ (Root mean square)
- Replace learning rate η by $\Delta\theta$ with $E[\Delta\theta^2]_t = \rho E[\Delta\theta^2]_{t-1} + (1 -$

RMSprop

- Similar to Adadelta, except that it still estimates η
- Compared with Adagrad, don't have to worry about decreasing step size, since there's a decay on it



$$\theta_{t+1} = \theta_t - \frac{\eta}{\mathrm{RMS}[g]_t} \odot \nabla J(\theta_t)$$

Adam

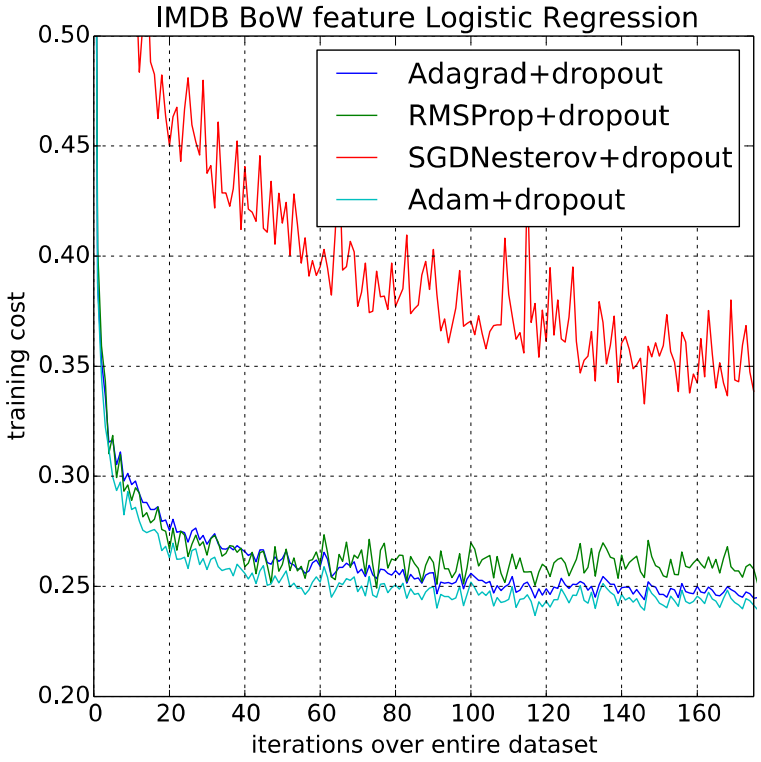
- Combine momentum and RMSProp: estimate both first and second moment

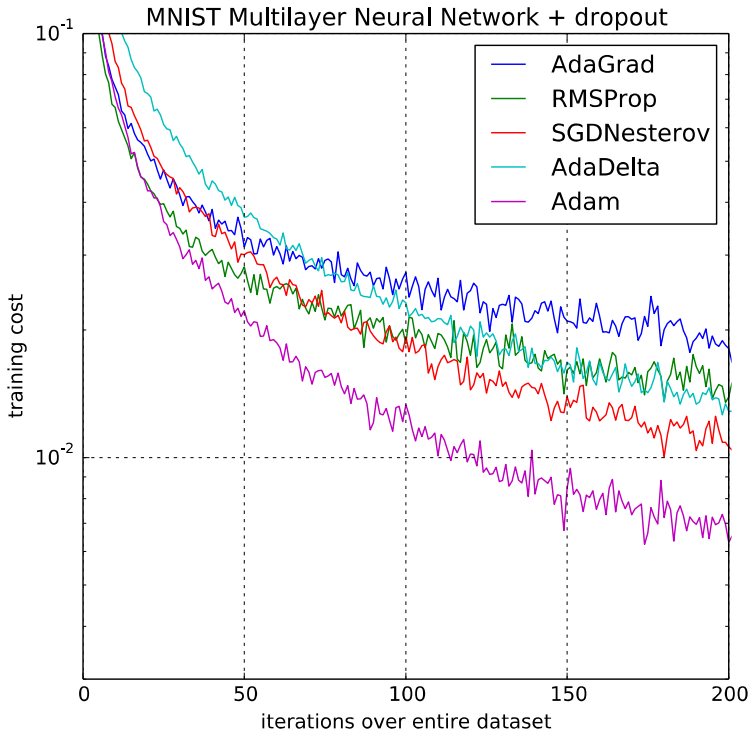
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

- Correct the bias of estimation when m, v are initialized at zero

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{ and } \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$


IMDB BoW feature Logistic Regression
Adagrad+dropout
RMSProp+dropout
SGDNesterov+dropout
Adam+dropout
training cost
iterations over entire dataset
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
20
40
60
80
100
120
140
160


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
iterations over entire dataset
10^{-1}
10^{-2}
0
50
100
150
200

Learning Rates in SGD

$$\theta_{t+1} = \theta_t - \eta_t g_t$$

where $g_t = g(\theta_t)$ is a stochastic gradient and the learning rate η satisfies

$$\sum_t \eta_t = \infty \text{ and } \sum_t \eta_t^2 < \infty$$

- Convergence of SGD is slow and it is difficult to tune the stepsize η_t

Adaptive Learning Rates

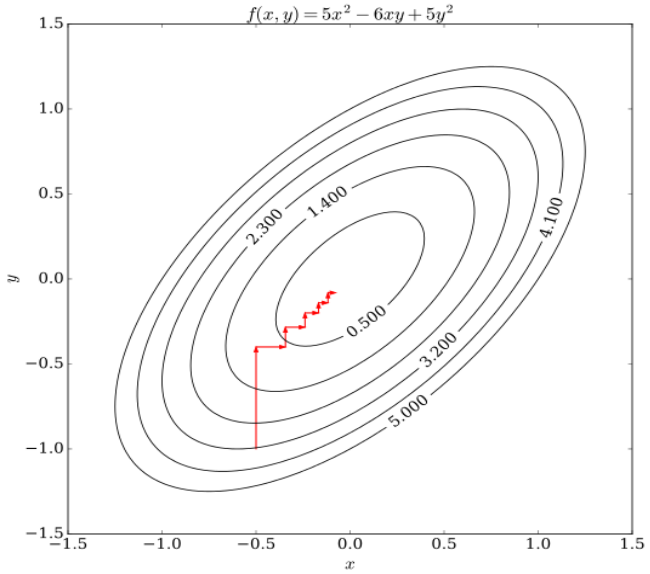
- Learning rates too large, diverge or flip around; too small, move slow and little progress
- In a multilayer neural net, the appropriate learning rates can vary widely between weights
- “One learning rate per group of parameters”
- Data-driven, as few hyper-parameters as possible

Other optimization strategies

- Block Coordinate Descent
- Supervised Pretraining
- Knowledge Distillation
- Continuation Method and Curriculum Learning

Block Coordinate Descent

- Update a block of parameters iteratively
- BCD is efficient when
  - the different variables in can be clearly separated
  - optimization w.r.t. the block variables is significantly easier

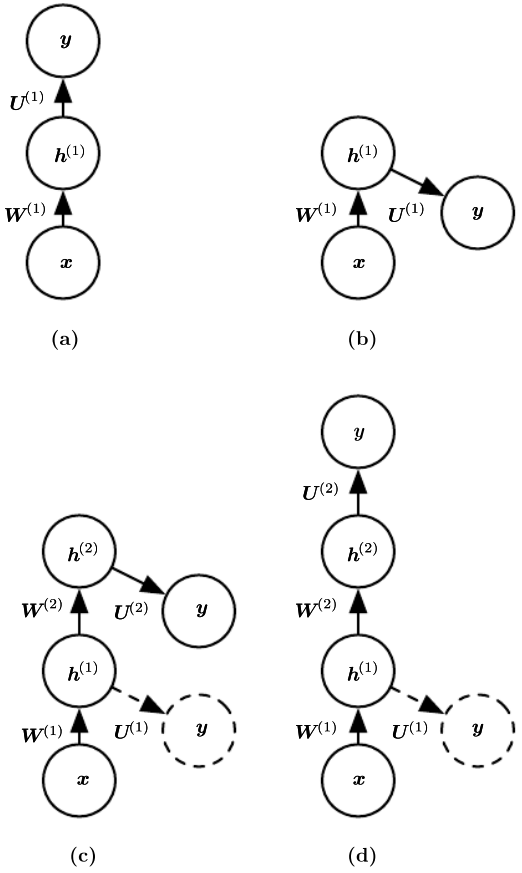


Supervised Pretraining

- Train simple models on simple tasks, then move on to more complex models and tasks.
- Greedy pretraining
  - Break a problem into many parts
  - Combine optimized components and fine tune the full model
  - No theoretical results, but works very well

Greedy Algorithms

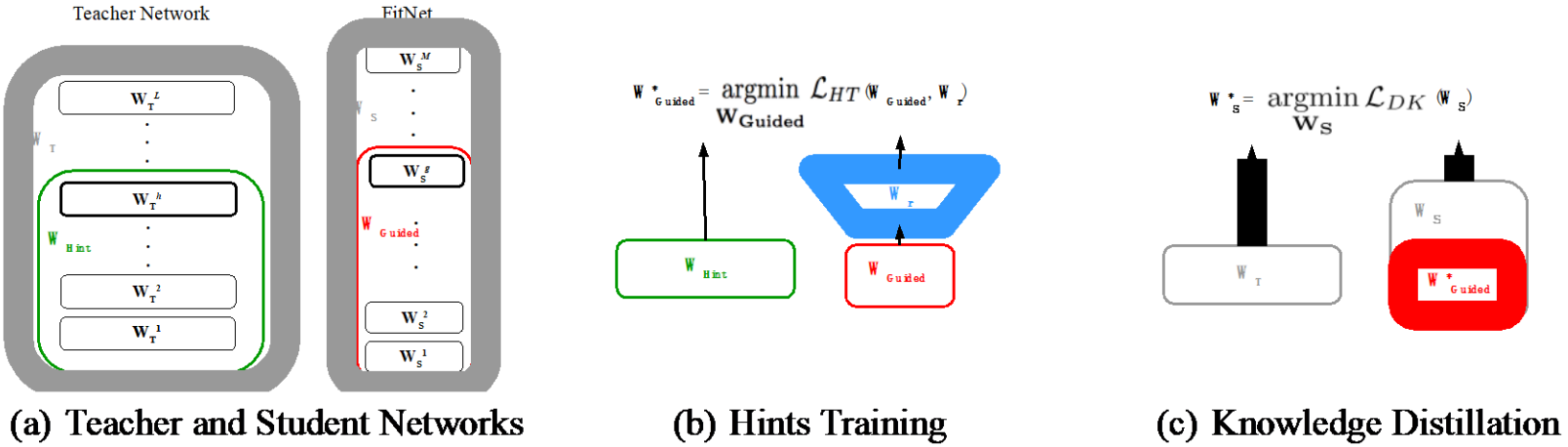
- Added hidden layer is pretrained as part of a shallow supervised MLP
- Take the output of the previously trained hidden layer as input.



Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007c). Greedy layer-wise training of deep networks. In *Adv. Neural Inf. Proc. Sys. 19* , pages 153–160. 275

Knowledge Distillation

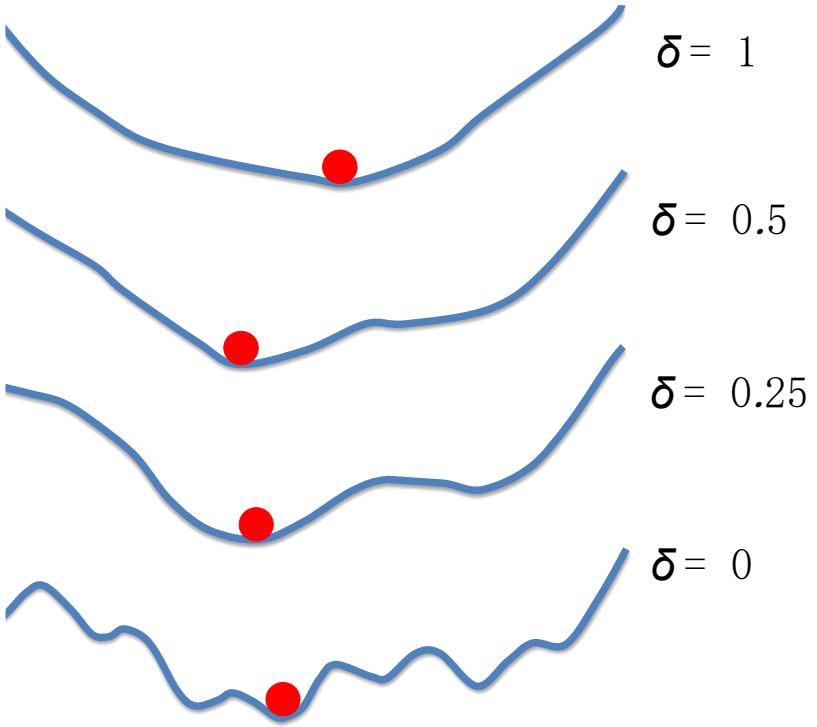
- Pretrain a *teacher* net to help improve *student* net's performance



Romero, A., Ballas, N., Ebrahimi Kahou, S., Chassang, A., Gatta, C., and Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *ICLR'2015, arXiv:1412.6550* . 277

Continuation Method and Curriculum Learning

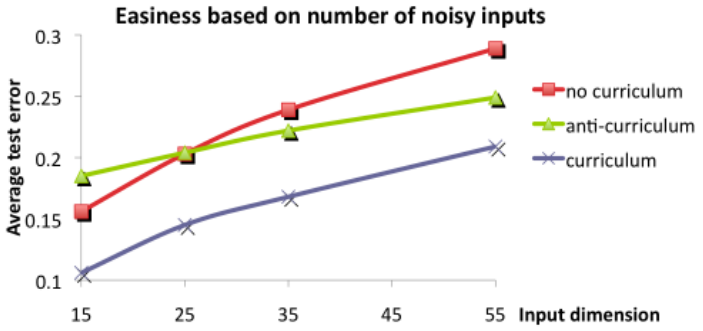
- Solve a series of problems with increasing difficulty
- Traditional CM based on smoothing, e.g. local averaging:
  $$\hat{f}_\delta(\boldsymbol{x}) = \mathrm{E}_{\boldsymbol{u}\sim B}[f(\boldsymbol{x} + \delta\boldsymbol{u})]$$
- Escape local minima
- Speed up convergence

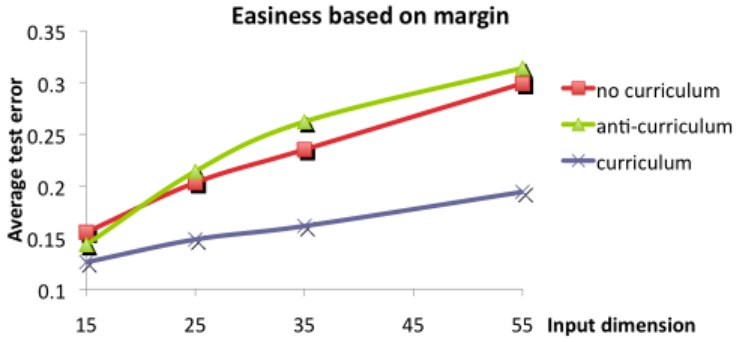


Hazan et al. 2015

Curriculum Learning

- Learning is more efficient when data are not i.i.d., but arranged in a meaningful order with increasing complexity





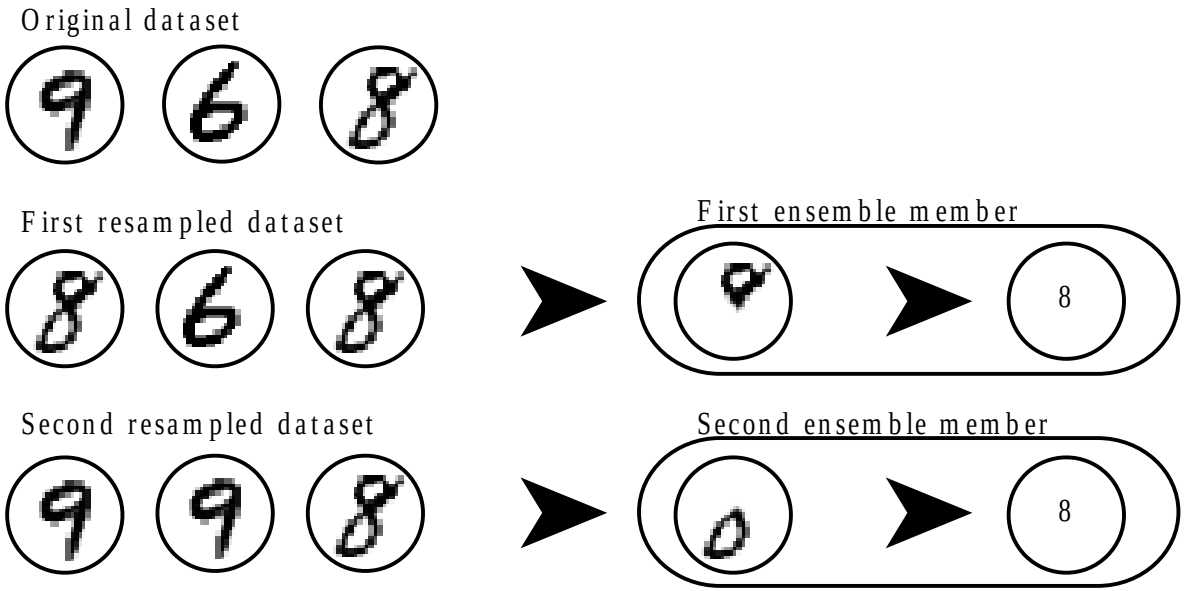
Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *ICML'09*. ACM. 279

Regularization in Neural Nets

Regularization in Neural Nets

Bagging and Ensemble Methods

- Train *k* different models separately
- Each of the *k* models resample the training data
- All the models vote on the output for test examples



Regularization in Neural Nets

- Regularizer and constraint models
- Dropout
- Data augmentation and others

Parameter Norm Penalty

$$\min_{\theta} \tilde{J}(\theta) = J(\theta; X, y) + \alpha\Omega(\theta)$$

$J(\theta; X, y)$ stands for the loss driven by data, not $l(\cdot)$.

- Regularizers are added to
  - encode some prior knowledge
  - express a generic preference for a *simpler* model and promote *generalization*

L^2 Parameter Regularization

- L^2 parameter norm penalty is known as *weight decay*.

$$\tilde{J}(w) = J(w; X, y) + \frac{\alpha}{2}\|w\|^2$$

- Gradient step

$$\begin{aligned} w &= w - \eta\left(\alpha w + \nabla_w J(w; X, y)\right) \\ &= (1 - \eta\alpha)w - \eta\nabla_w J(w; X, y) \end{aligned}$$

L^2 penalty performs weight shrinkage by a factor $(1 - \eta\alpha)$.

L^2 Penalty on Quadratic Problem

Ridge Regression

$$\min_w \tilde{J}(w) = \frac{1}{2}\|y - Xw\|^2 + \frac{\alpha}{2}\|w\|^2$$

where $w^* \in \operatorname{argmin}_w J(w)$ and $J(w) = \frac{1}{2}\|y - Xw\|^2$ being the original QP. Then

$$w^* = (X^T X)^{-1} X^T y$$

Let $\nabla \tilde{J}(\tilde{w}) = 0$, we have

$$\tilde{w} = (X^T X + \alpha I)^{-1} X^T y$$

L^2 Penalty on Quadratic Problem

- SVD: $X = U\Sigma V^T$ (Assume X full column rank)

  $X^T X = \mathrm{V}\Sigma^2 V^T$ and $X^T X + \alpha I = \mathrm{V}(\Sigma^2)V^T$

- Prediction:

$$y^* = Xw^* = UU^T y$$
$$\tilde{y} = X\,\tilde{w} = U\Sigma(\Sigma^2 + \alpha I)^{-1}\Sigma U^T y$$

Equivalently:

$$\tilde{y} = \sum_i \frac{\sigma_i^2}{\sigma_i^2 + \alpha} u_i (u_i^T y)$$

Intuition

- $X^T X = V \Sigma^2 V^T$ is the scaled covariance
- Columns of V are principle directions (PCA)
- Columns of U: representation of data in the principle direction

$$\sigma_i u_i = X v_i$$

- In ridge regression: $\tilde{y} = \sum_i \frac{\sigma_i^2}{\sigma_i^2 + \alpha} u_i (u_i^T y)$
  - shrinkage on all the σ_is, but small σ_i are affected mostly.
- Ridge regression respects and assigns higher weights to the more informative directions, where most of the data activity take place

L_1 Parameter Regularization

$$\tilde{J}(w) = J(w; X, y) + \alpha \|w\|_1$$

- $\nabla_w \tilde{J} = \alpha \text{sign}(w) + \nabla_w J(w)$, however not continuously differentiable
- Consider a simple case

$$J(w) = \frac{1}{2}(w - w^*)^T \text{diag}(h)(w - w^*)$$

it can be shown that at optimality, we have

$$w_i = \text{sign}(w_i^*) \max\{|w_i^*| - \frac{\alpha}{H_{i,i}}, 0\}$$

- The so-called *soft-thresholding operator* either shifts w_i^* or truncates it to zero.

Norm Penalties as Constrained Optimization

$$\min_{\theta} \quad \tilde{J}(\theta) = J(\theta; X, y)$$
$$\text{s.t.} \quad \Omega(\theta) \leq k$$

- Lagrange function $\mathcal{L}(\theta, \alpha) = J(\theta; X, y) + \alpha(\Omega(\theta) -$

Norm Penalties as Constrained Optimization

- More specifically, we have a correspondence between α and k such that solving the regularized problem and constrained problem yield the same optimal θ.
- The penalty α doesn't give explicit form of k, which is often needed to characterize the constraint region.
- We can tweak the magnitude of α to find a path of solutions
  - Large α-> small region
  - Small α-> large region

Reasons to Use Explicit Constraints

- When we have an idea of what value of k is appropriate
- Regularizer will encourage optimizer to get stuck in local minima corresponding to small $\|\theta\|$
- Explicit constraints may prevent divergence when using high learning rates.

Reasons to Use Regularizers

- Improve stability, handle ill-conditioned problem
- Often the corresponding optimization is easier
- Quite useful in generalized linear models (shallow model)

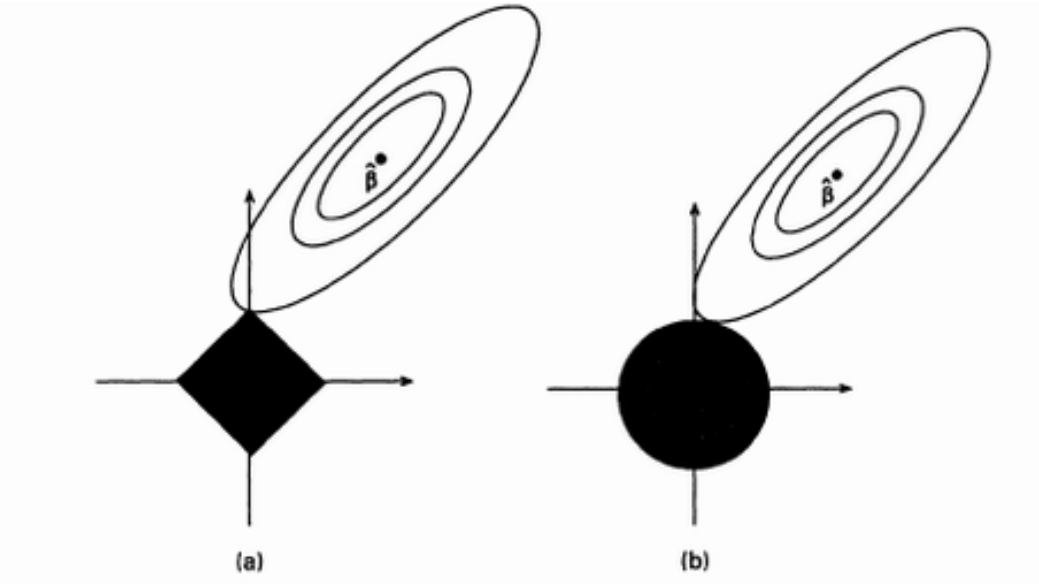
Difference in L_2 and L_1 Penalty

- L^2 penalize all the parameters and drive them to zero
- L_1 performs truncation and enhance sparse solution

$$\hat{\beta}^* = argmin_{x \in \mathbb{R}^n} f(x)$$

$$\min_{\beta} f(\beta)$$
$$\text{s.t. } \|\beta\|_k \leq \lambda$$

where $k = 1,2$



Sparse Representations

- Besides, in deep learning, we often invoke sparse penalty on *representation*.

$$\min_{\theta} \tilde{J}(\theta) = J(\theta; X, y) + \alpha\Omega(h)$$

where $\Omega(h)$ is the L_1 or KL divergence to constrain the neurons' activity.

Dropout: Training

- Use a mask vector μ to randomly turn on/off each neuron
- To apply SGD, we not only sample mini-batch of data, but sample the neurons as well.

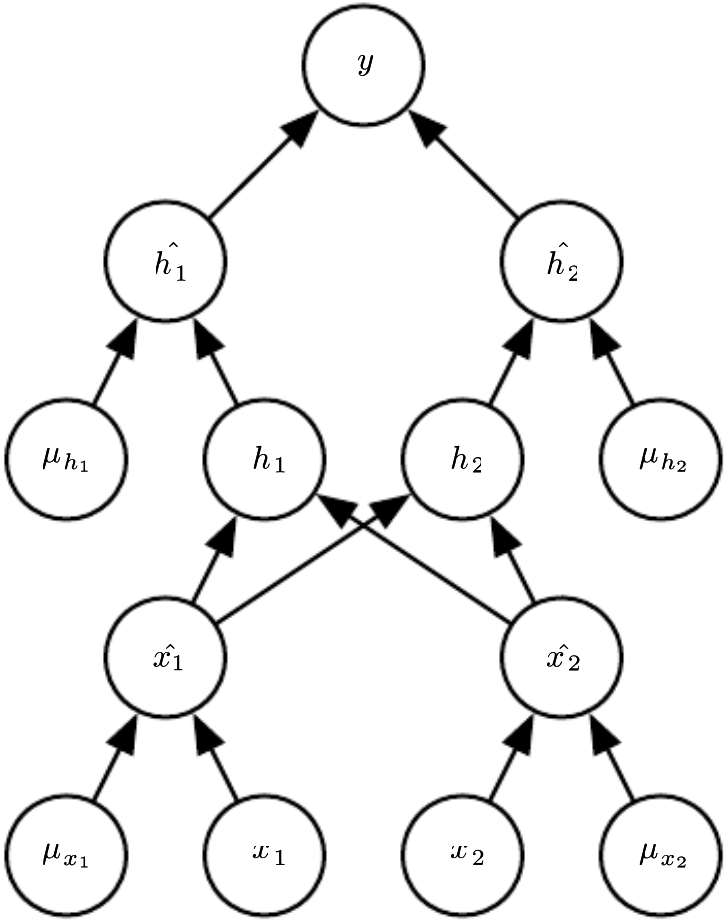
for $t = 1,2,3, \ldots$

sample $i \in \{1,2, \ldots, n\}$ and $\boldsymbol{\mu} \in 2^d$

$$\mathbf{s} = \nabla l\left(f\left(\mathrm{x}^{(i)}, \boldsymbol{\theta}_{\boldsymbol{\mu}}^{(t)}, \mu\right), y^{(i)}\right) + \lambda \nabla \Omega\left(\boldsymbol{\theta}_{\boldsymbol{\mu}}^{(t)}\right)$$

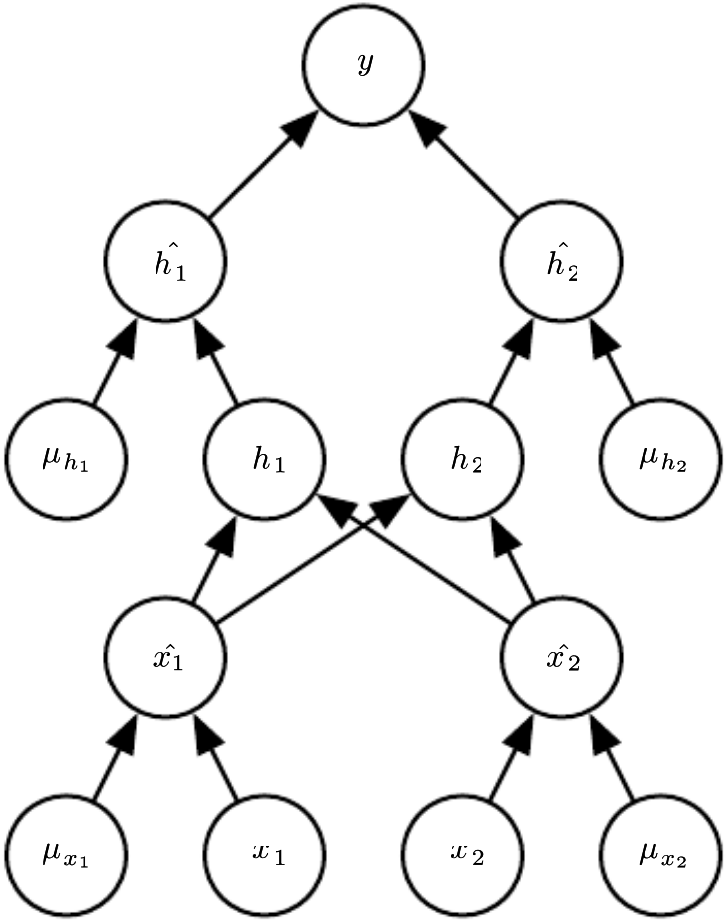
$$\boldsymbol{\theta}_{\boldsymbol{\mu}}^{(t+1)} = \boldsymbol{\theta}_{\boldsymbol{\mu}}^{(t)} - \eta \cdot \mathbf{s}$$

$$\boldsymbol{\theta}_{\boldsymbol{\mu}^c}^{(t+1)} = \boldsymbol{\theta}_{\boldsymbol{\mu}^c}^{(t)}$$



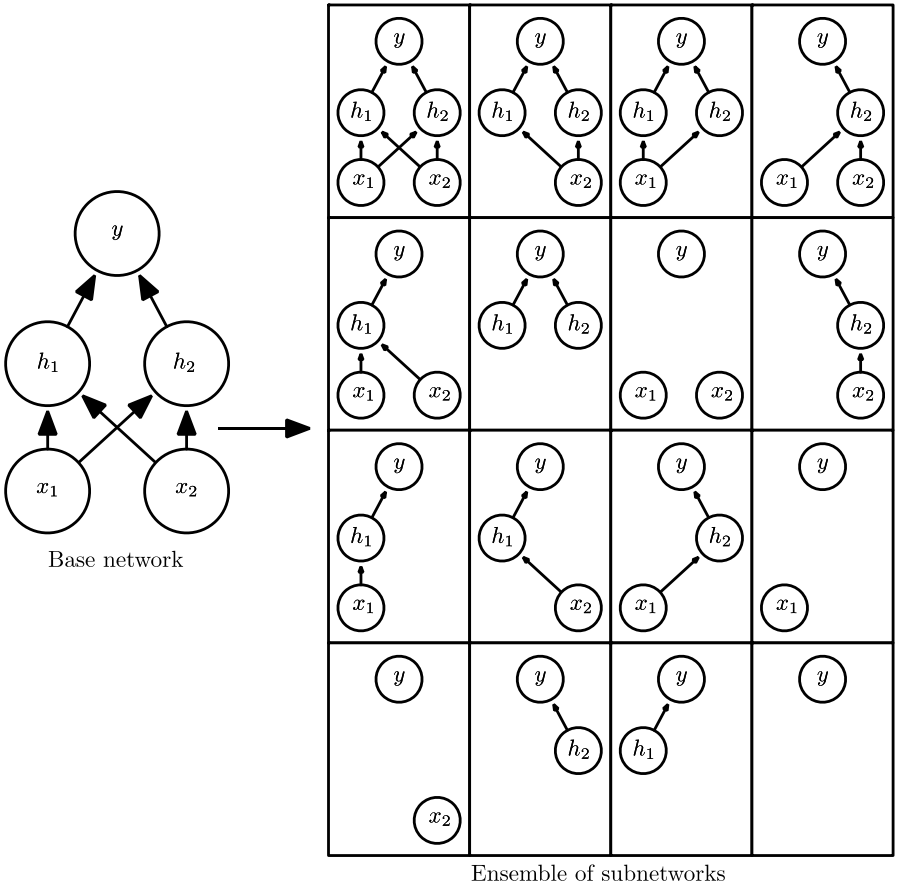
Dropout: Training

```
prob = .5

# forward prop
H1 = np.tanh(np.dot(W1, X) + b1)
mu1 = np.random.rand(*H1.shape) < prob
H1 *= mu1

out = np.tanh(np.dot(W2, H1) + b2)
mu2 = np.random.rand(*out.shape) < prob
out *= mu2
```



Dropout (Efficient Ensemble of NNs)

- Dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from an underlying base network
- We have exponentially many models (2^d)



Dropout: Prediction

- In Bagging, average the k models: $\frac{1}{k}\sum_i p^{(i)}(y|x)$
- In Dropout:

$$\sum_{\mu} p(\mu)p(y|x,\mu)$$

Although this is intractable, we can approximate it with finite sum.

Dropout: Prediction

- Weight scaling inference rule:
  - Evaluate only one specific model
  - the output of units are multiplied by the probability of turning on: $\mathbf{h}' = \mathbb{E}[\boldsymbol{\mu}] \odot \mathbf{h}$
- Example:
  - $y = h_1 + h_2$
  - In training: $\mathbb{E}[\mathbf{y}] = \mathbf{p2}(\mathbf{h_1} + \mathbf{h2}) + \mathbf{p}(\mathbf{1} - \mathbf{p})\mathbf{h_1} + \boldsymbol{p}(\mathbf{1} - \boldsymbol{p})\boldsymbol{h_2} + (\mathbf{1} - \boldsymbol{p})^{\mathbf{2}} * \mathbf{0} = \boldsymbol{p}(\mathbf{h_1} + \mathbf{h2})$
  - So during testing: $\mathbf{y} = \boldsymbol{p}(\mathbf{h_1} + \mathbf{h2})$

More common: Inverted dropout

```
# forward prop during train
prob = .5
H1 = np.tanh(np.dot(W1, X) + b1)
mu1 = (np.random.rand(*H1.shape) < prob) / prob
H1 *= mu1

out = np.tanh(np.dot(W2, H1) + b2)
mu2 = (np.random.rand(*out.shape) < prob) / prob
out *= mu2

...

# during prediction
H1 = np.tanh(np.dot(W1, X) + b1)
out = np.tanh(np.dot(W2, H1) + b2)
```

Dropout: Prediction

- A better way is to use *geometric mean*, which approximates the predictions of the entire ensembles

$$\tilde{p}(y|x) = \sqrt[2^d]{\Pi_\mu p(y|x,\mu)}$$

Dropout: randomness at train time

- When training
  - Add randomness
- When testing
  - Average out randomness

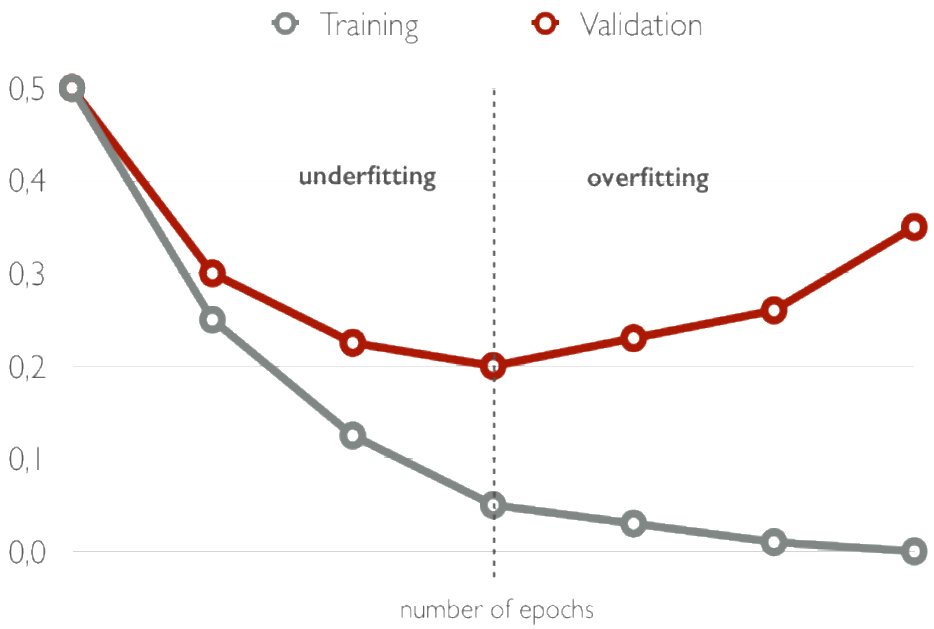
- Other example
  - Batch Normalization

Other Ways to Improve Generalization

- Data augmentation
  - Create new fake data by transforming the input features.
  - This works successfully for classification problems, such as object recognition.
- Random noise
- Semi-supervised Learning
  - Use unlabeled data to learn representation of feature x.
- Multitask Learning

Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



Train Very Deep Convolutional Nets

- VGGNET
- Pretrain a ConvNet
- Use the first four and last three layers to initialize a even deeper net

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

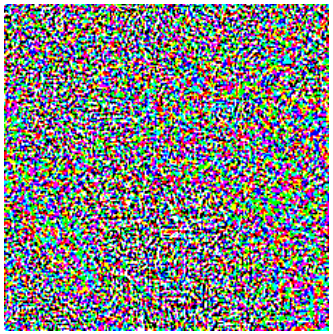
Supervised pretraining in deep ConvNets (Simonyan & Zisserman 2015)

Adversarial Training

- A neural network which performs well on human level accuracy would be entirely wrong on examples that are intentionally constructed (adversarial examples).

Adversarial Training

$+ .007 \times$



$=$

$\boldsymbol{x}$

$y =$“panda”
w/ 57.7%
confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

“nematode”
w/ 8.2%
confidence

$\boldsymbol{x} +$
$\epsilon \, \text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

“gibbon”
w/ 99.3 %
confidence

Summary

- Optimization
  - Initialization
  - SGD algorithms
  - Data preprocessing
  - …
- Regularization
  - Norm regularizations
  - Dropout
  - Bagging and ensembles
  - …