CMC MSU Department of Algorithmic Languages
Samsung Moscow Research Center

Neural Networks for Natural Language Processing

Нейронные сети в задачах автоматической обработки текстов

Lecture 8. Tips and tricks.

Arefyev Nikolay
CMC MSU Department of Algorithmic Languages &
Samsung Moscow Research Center

Plan

- **Data normalization, weights initialization.**
- Advanced optimization
- Cross-validation

Backprop reminder

- Define error as $\delta_j^{(l)} = \frac{\partial e(w)}{\partial z_j^{(l)}}$
- Calculate for the last layer
  - for sigmoid + CE
    y is 0/1
    $\delta_1^{(L)} = \hat{y} - y$
  - similar for softmax + CE
    y is one-hot vector
    $\delta^{(L)} = \hat{y} - y$

Predicted-real output looks like error!

- Recurrently calculate error in (l-1)-th layer from error in l-th layer

$$\delta^{(l-1)} = W^{(l)T} \delta^{(l)} * g'(z^{(l-1)})$$

Backpropagate error

- Calculate gradient w.r.t. weights from errors

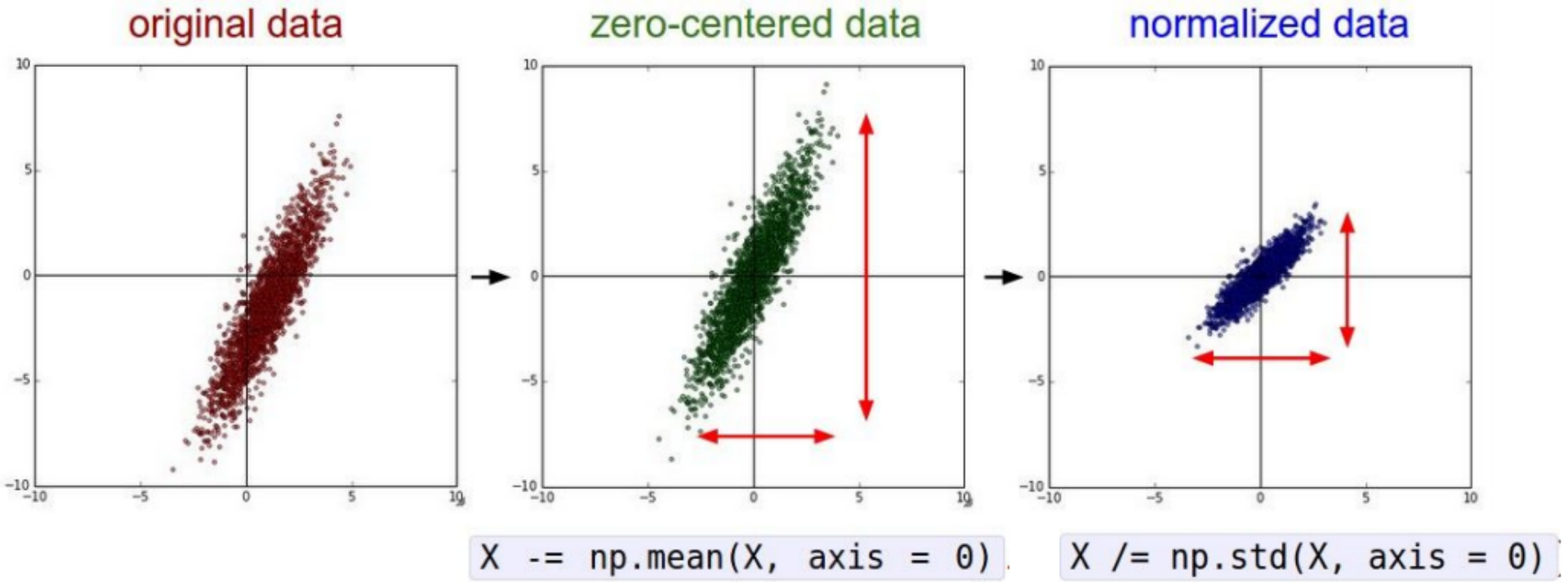
  For efficiency:

$$\frac{\partial e}{\partial W^{(l)}} = \delta^{(l)T} a^{(l-1)}$$

  - In forward prop save intermediate quantities required by backprop!
  - Use matrices D, Z, A of shape (batch_size, layer_size) and adapt formulas, always keep batch_size as first axis (common practice)

Data preprocessing

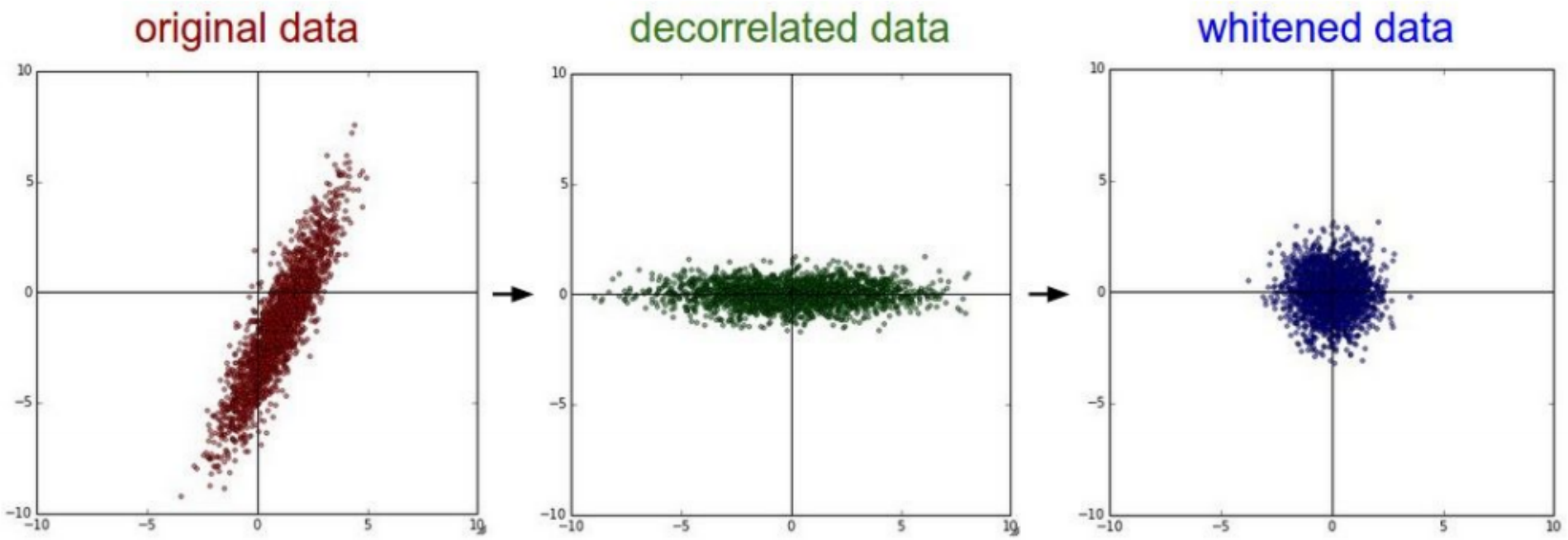
- Zero mean ←WHY?
  - Calculate mean for each feature on **train set**. Subtract from all examples (including test).
  - Not common for NLP: 0 occurences!
- Unit variance / equal feature scales ← WHY?



Fei Fei Li, Anrej Karpathy, Justin Johnson. Cs231n slides, 2016 @ http://cs231n.stanford.edu

Data preprocessing

- Ideally we want whitened data. But it's very computationally expensive.
- Converting features to the same scale is usually enough!



Fei Fei Li, Anrej Karpathy, Justin Johnson. Cs231n slides, 2016 @ http://cs231n.stanford.edu

Initializing FFNN weights

$$w_{ij}^{(l)}(t) := w_{ij}^{(l)}(t-1) - \lambda \delta_j^{(l)} a_i^{(l-1)}$$

If all weights initialized to 0s (or other constant):

- all activations in the previous layer will compute the same function (will be equal)
- all errors backpropagated from the next layer will be equal ← WHY?

all updates will be the same and all weights will stay the same. Similar to using hidden layers of size 1!

- Break symmetry: initialize weights with small random numbers
  - 0.01*np.random.randn(s_l, s_l-1) are ok for small NNs

    from special distributions (Xavier/Glorot/He) to keep activations & gradients from vanishing or exploding.

Init FFNNs with small random numbers

10 layers
500 neurons each
tanh activatations
w~randn()*0.01

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

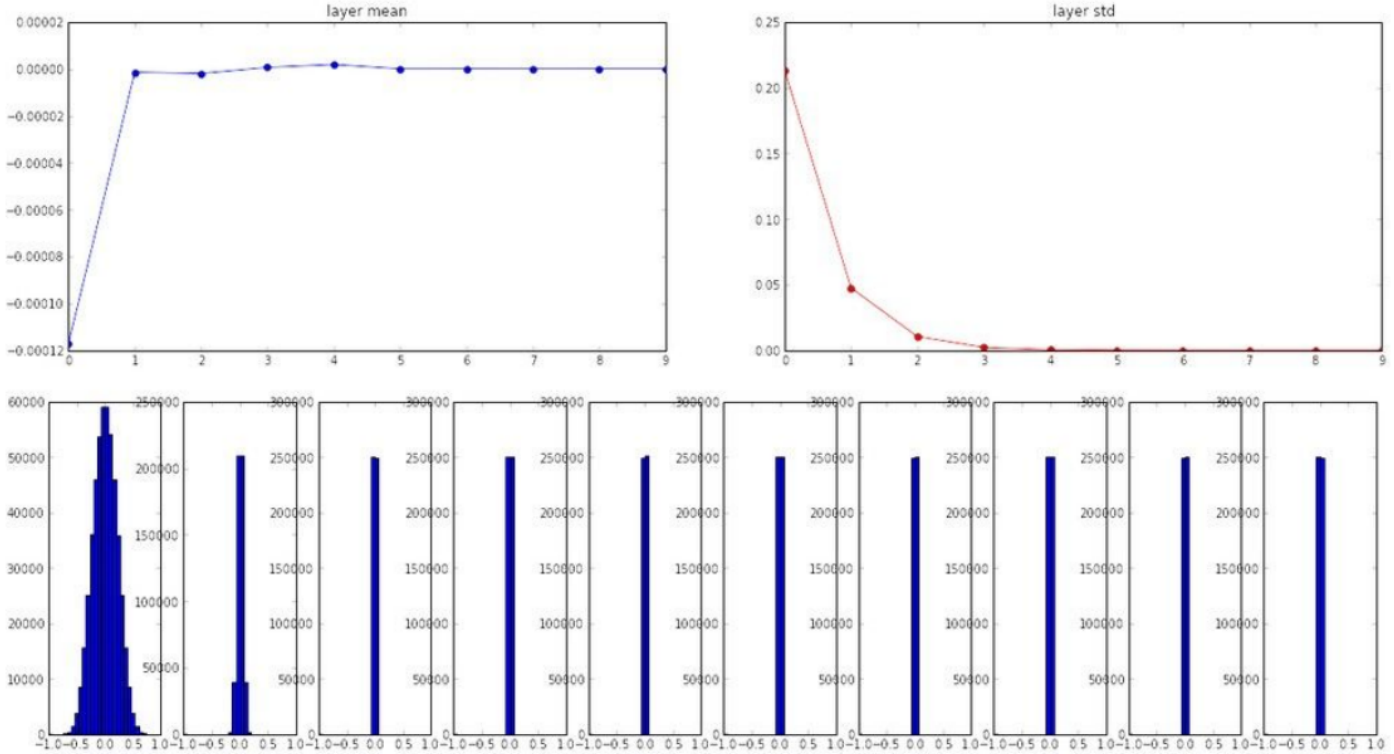
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Init FFNNs with small random numbers

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

Q: think about the backward pass. What do the gradients look like?

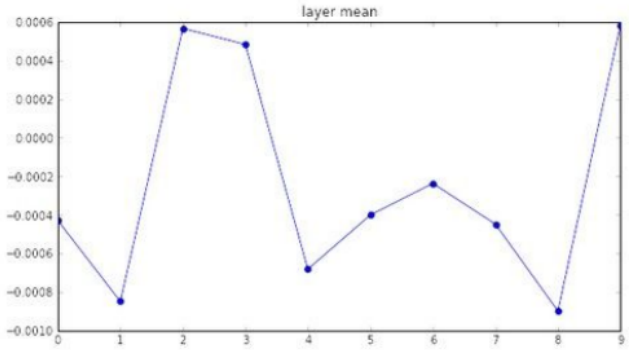
Hint: think about backward pass for a W*X gate.

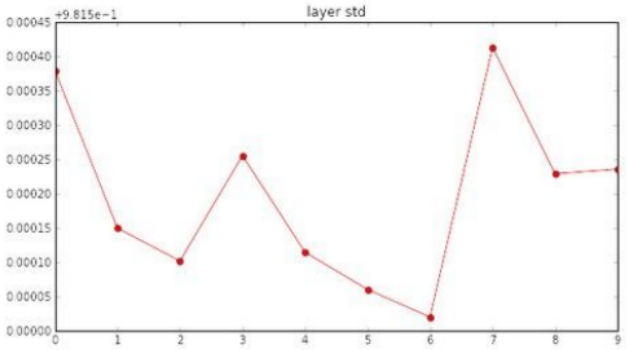
Init FFNNs with large random numbers

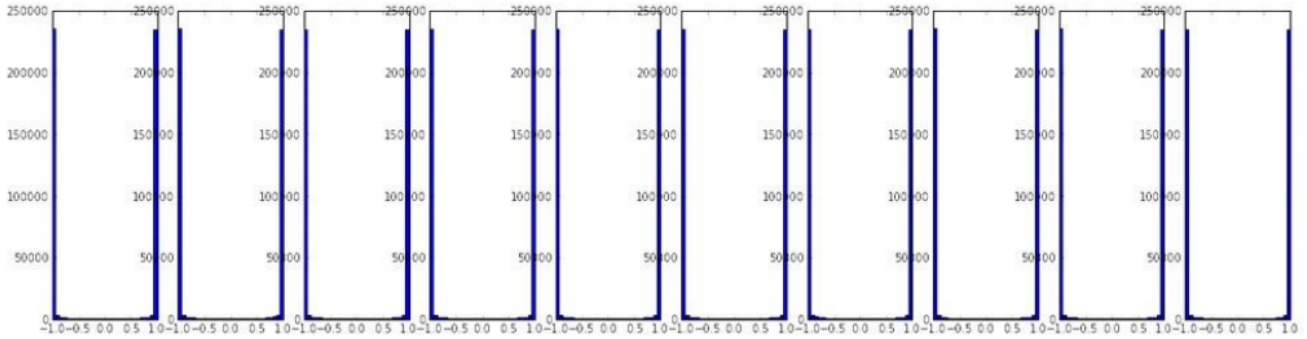
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01







Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Init FFNNs principle

If inputs have zero mean and unit variance, activations in each layer should have them also!

After random init:

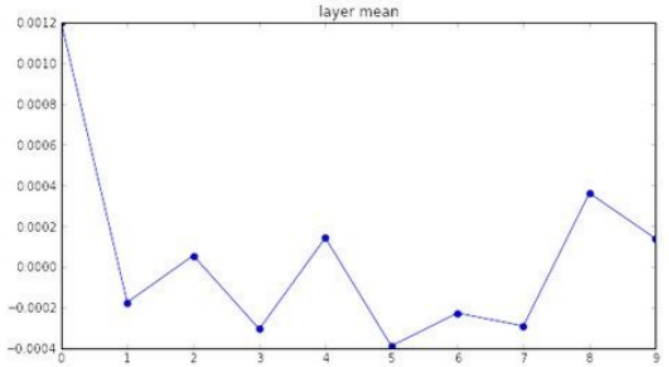
- Var(wx) = fan_in Var(w) Var(x) ←DERIVE
  - Init with Var(w)=1/fan_in to save variance of the input
  - Additional corrections for saving variance in activations and in backprop

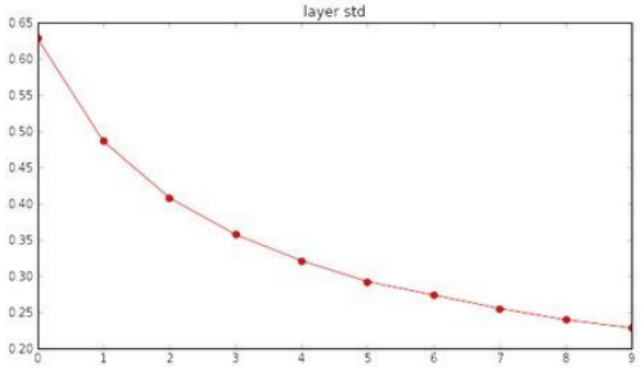
Init FFNNs with Xavier init

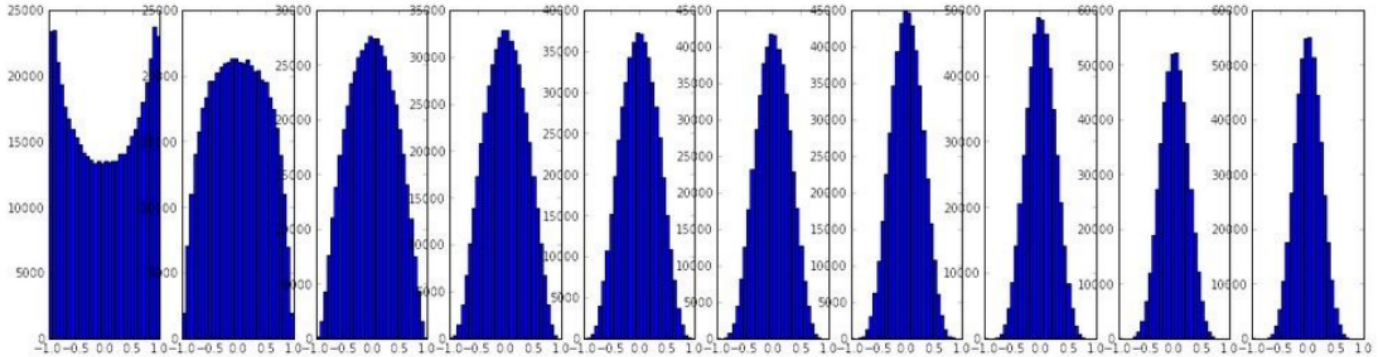
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]







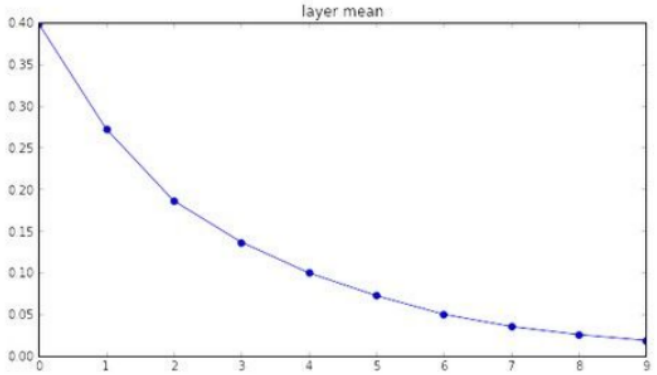
Reasonable initialization.
(Mathematical derivation assumes linear activations)

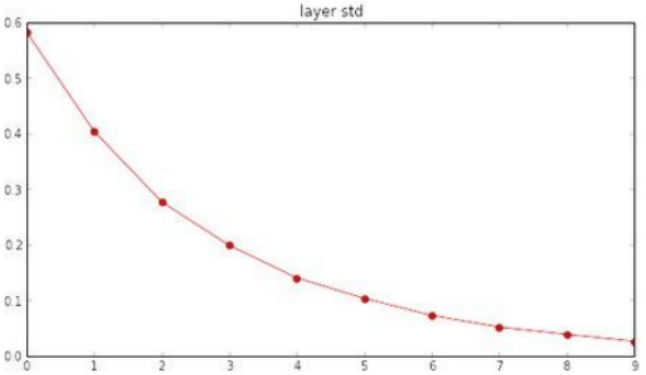
Xavier init fails for ReLU

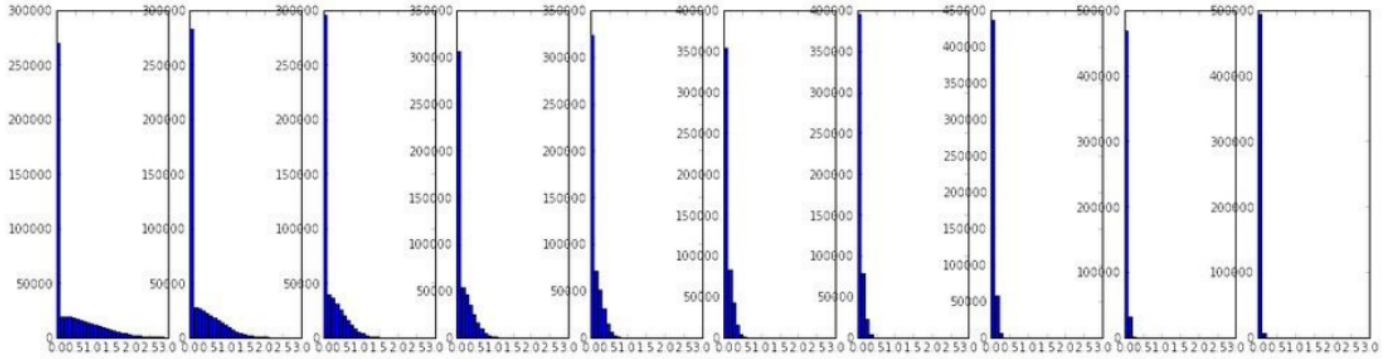
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.





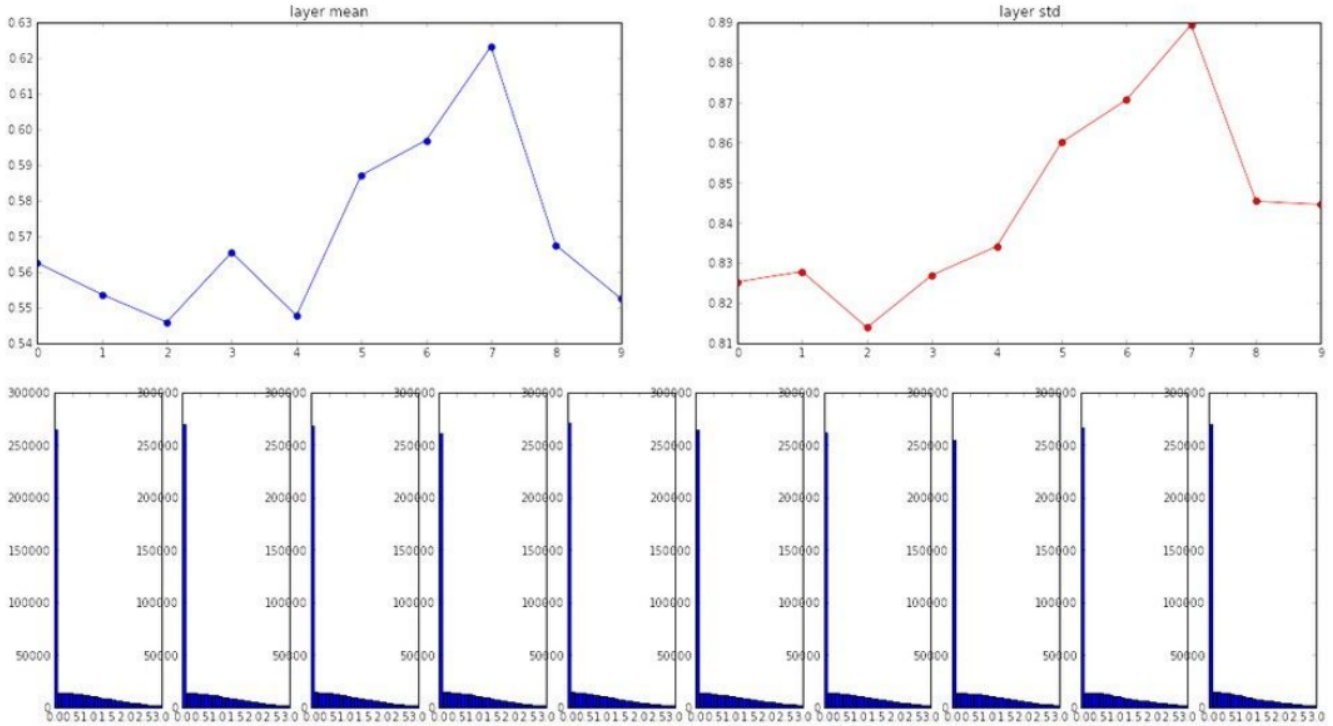


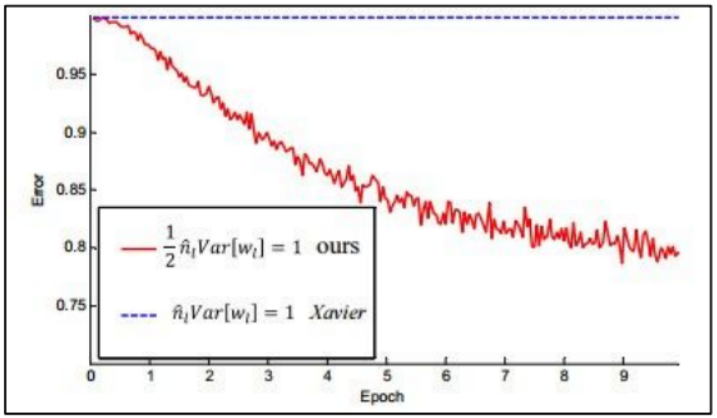
Init FFNNs with He init

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(note additional /2)



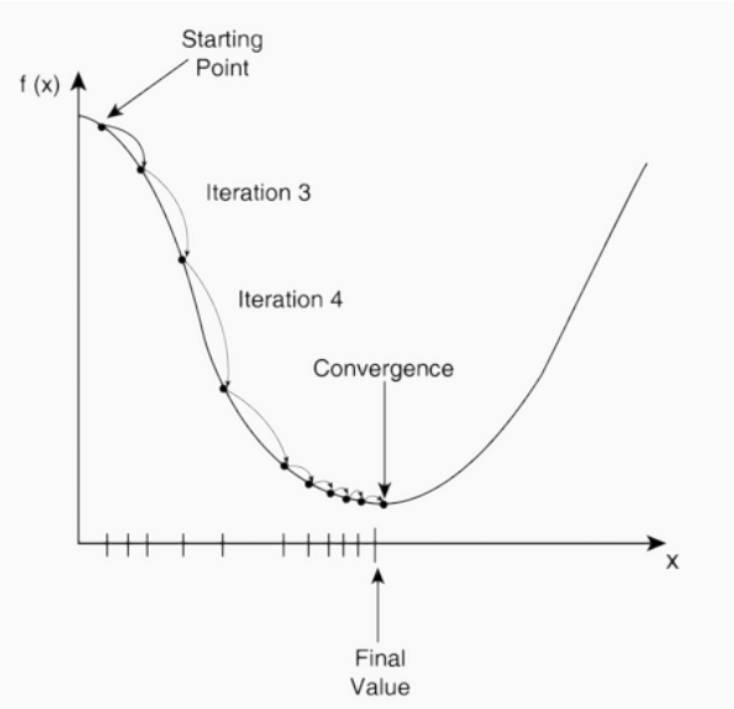


Plan

- Data normalization, weights initialization.
- **Advanced optimization**
- Cross-validation

SGD

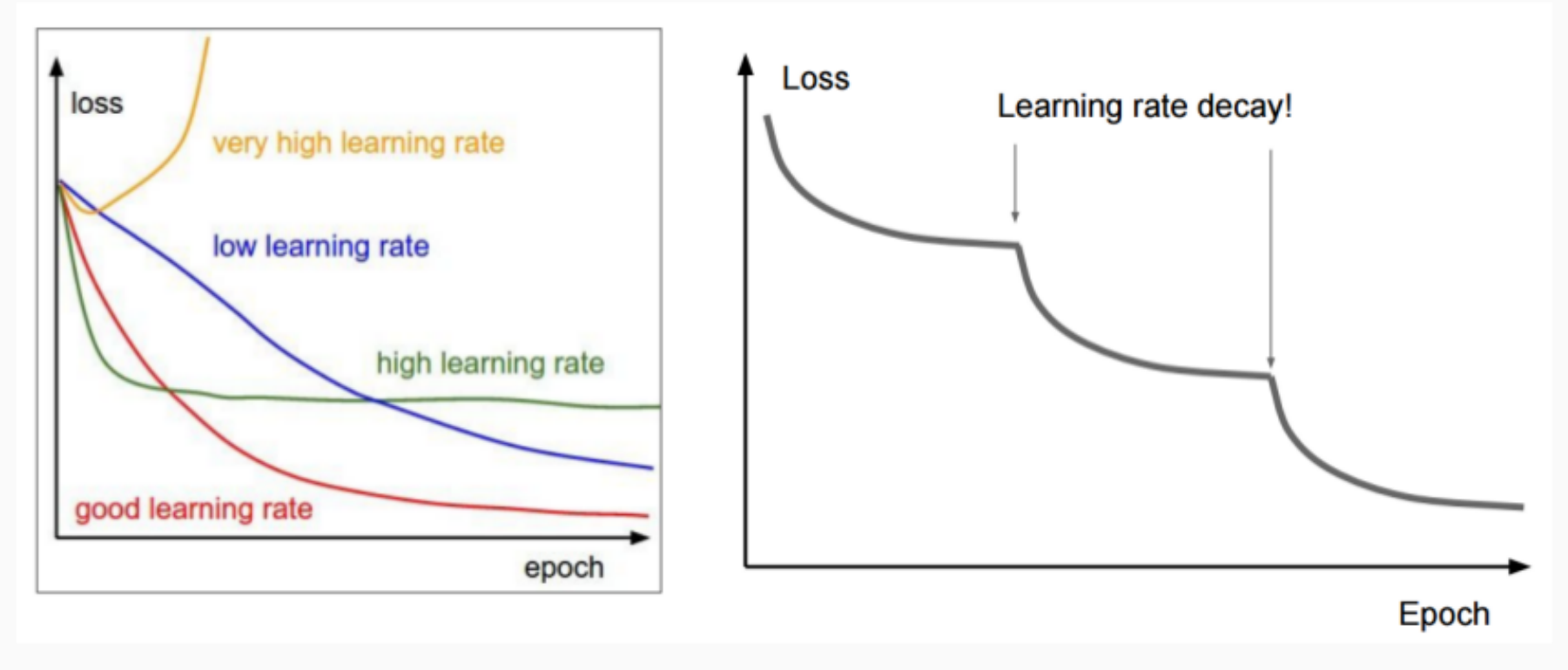
- Gradient descent is a way to minimize an objective function $J(\theta)$
  - $\theta \in \mathbb{R}^d$: model parameters
  - η: learning rate
  - $\nabla_\theta J(\theta)$: gradient of the objective function with regard to the parameters
- Updates parameters **in opposite direction** of gradient.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$



Sebastian Ruder. Optimization for Deep learning, 2017

Learning rate

- Learning rate – one of the most important hyperparameters!



(Full) batch gradient descent

- Computes gradient with the **entire** dataset.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(
    loss_function, data, params)
  params = params - learning_rate * params_grad
```

Listing 1: Code for batch gradient descent update

Sebastian Ruder. Optimization for Deep learning, 2017

(Full) batch gradient descent

- Computes gradient with the **entire** dataset.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(
    loss_function, data, params)
  params = params - learning_rate * params_grad
```

Listing 1: Code for batch gradient descent update

- **Very slow**.
- Intractable for datasets that **do not fit in memory**.
- **No online learning**.

Sebastian Ruder. Optimization for Deep learning, 2017

Stochastic gradient descent (SGD)

- Computes update for **each** example $x^{(i)}y^{(i)}$.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(
      loss_function, example, params)
    params = params - learning_rate * params_grad
```

- Pros
  - **Much faster** than batch gradient descent.
  - Allows **online learning**.
- Cons
  - **High variance** updates.

Sebastian Ruder. Optimization for Deep learning, 2017

Stochastic gradient descent (SGD)

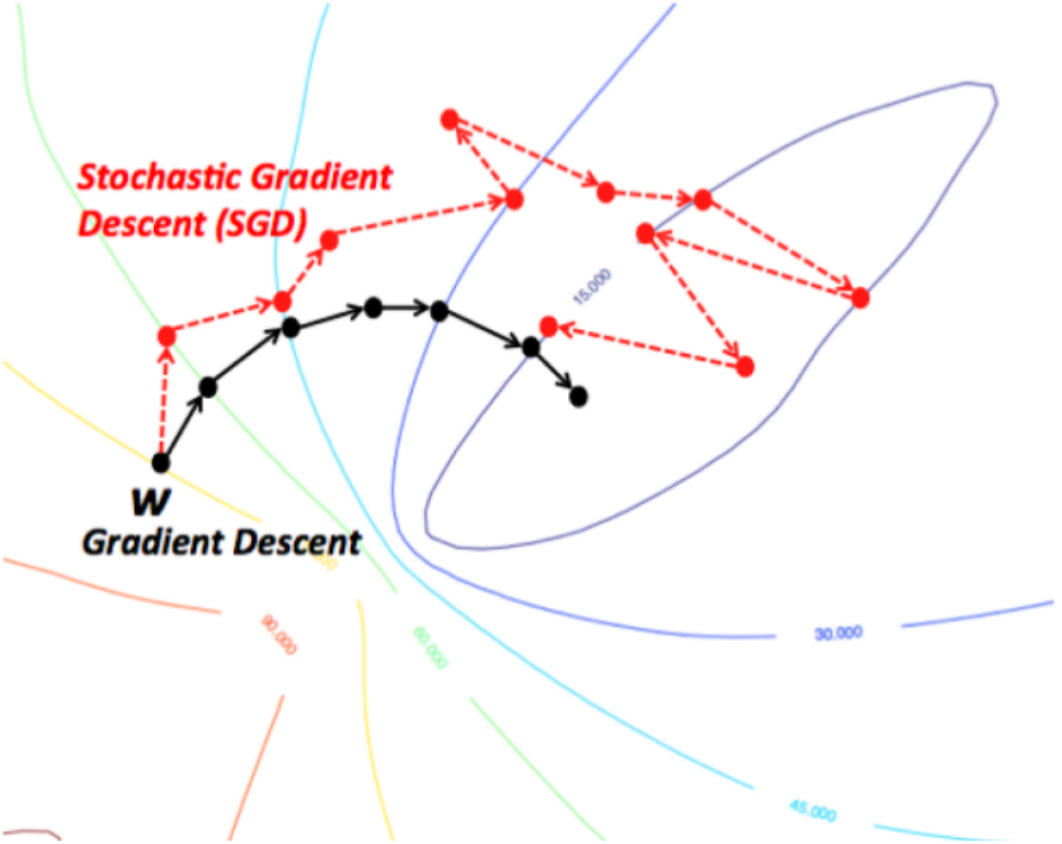


Figure: Batch gradient descent vs. SGD fluctuation (Source: wikidocs.net)

- SGD shows same convergence behaviour as batch gradient descent if learning rate is **slowly decreased (annealed)** over time.

Sebastian Ruder. Optimization for Deep learning, 2017

Mini-batch gradient descent

- Performs update for every **mini-batch** of n examples.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(
      loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Listing 3: Code for mini-batch gradient descent update

Sebastian Ruder. Optimization for Deep learning, 2017

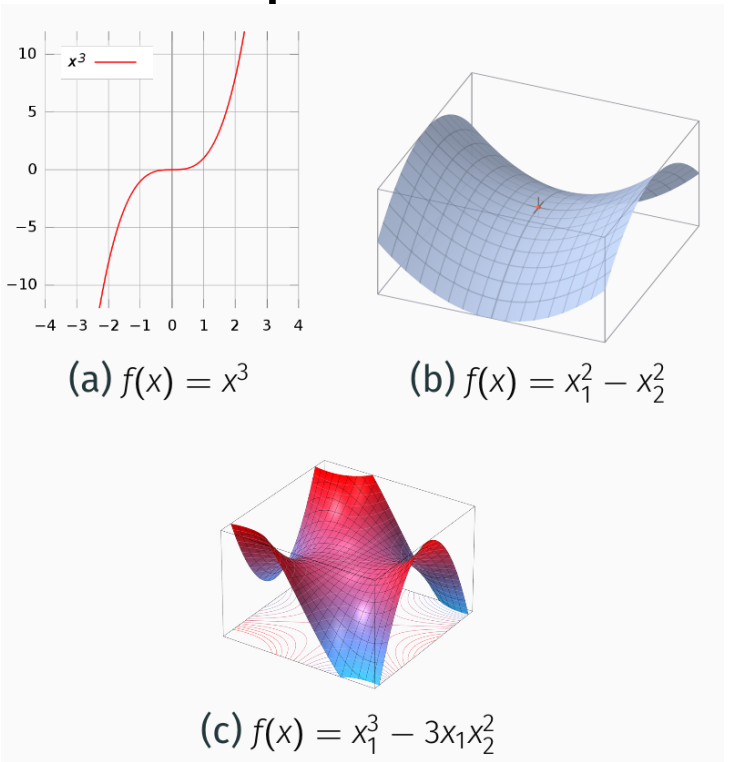
Mini-batch gradient descent

- Pros
  - **Reduces variance** of updates.
  - Can exploit **matrix multiplication** primitives.
- Cons
  - **Mini-batch size** is a hyperparameter. Common sizes are 50-256.
- Typically the algorithm of choice.
- Usually referred to as SGD even when mini-batches are used.

Sebastian Ruder. Optimization for Deep learning, 2017

Many “bad” local minima?

- Not for real NN losses in N-d!
  - Loss in most local minima is not much worse than in global minimum
  - In N-d there critical points are mostly saddle points



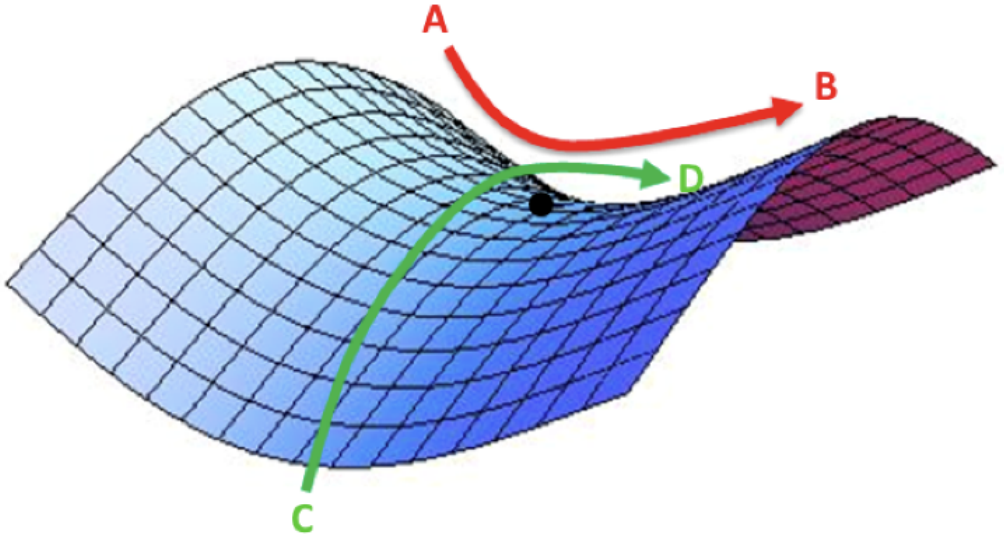
(a) $f(x) = x^3$

(b) $f(x) = x_1^2 - x_2^2$

(c) $f(x) = x_1^3 - 3x_1x_2^2$

Bad saddle points!

- Low probability SGD-based algorithms stuck in saddle points
- But they significantly slow down near saddle points (flat areas)

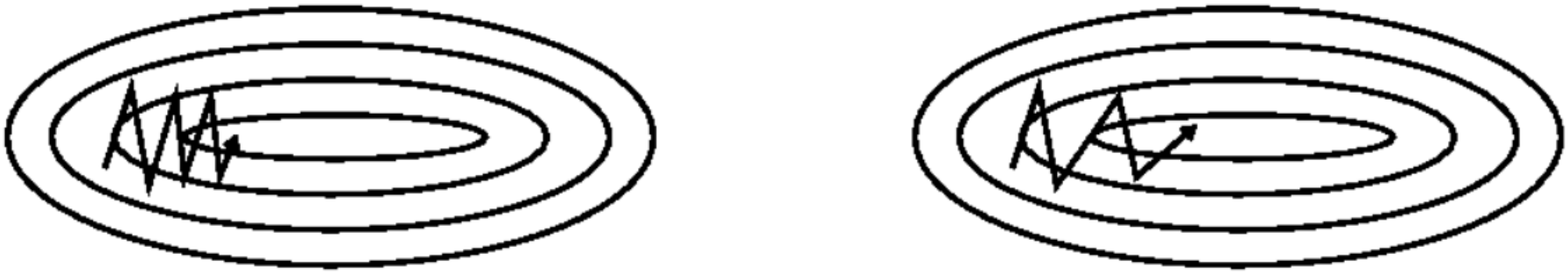


https://www.safaribooksonline.com/library/view/fundamentals-of-deep/9781491925607/ch04.html

Momentum

- SGD has trouble navigating **ravines**.
- Momentum [Qian, 1999] helps SGD **accelerate**.
- Adds a fraction γ of the update vector of the past step v_{t-1} to current update vector v_t. Momentum term γ is usually set to 0.9.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$



(a) SGD without momentum

(b) SGD with momentum

Figure: Source: Genevieve B. Orr

Sebastian Ruder. Optimization for Deep learning, 2017

Momentum

- **Reduces updates** for dimensions whose gradients **change directions**.
- **Increases updates** for dimensions whose gradients **point in the same directions**.

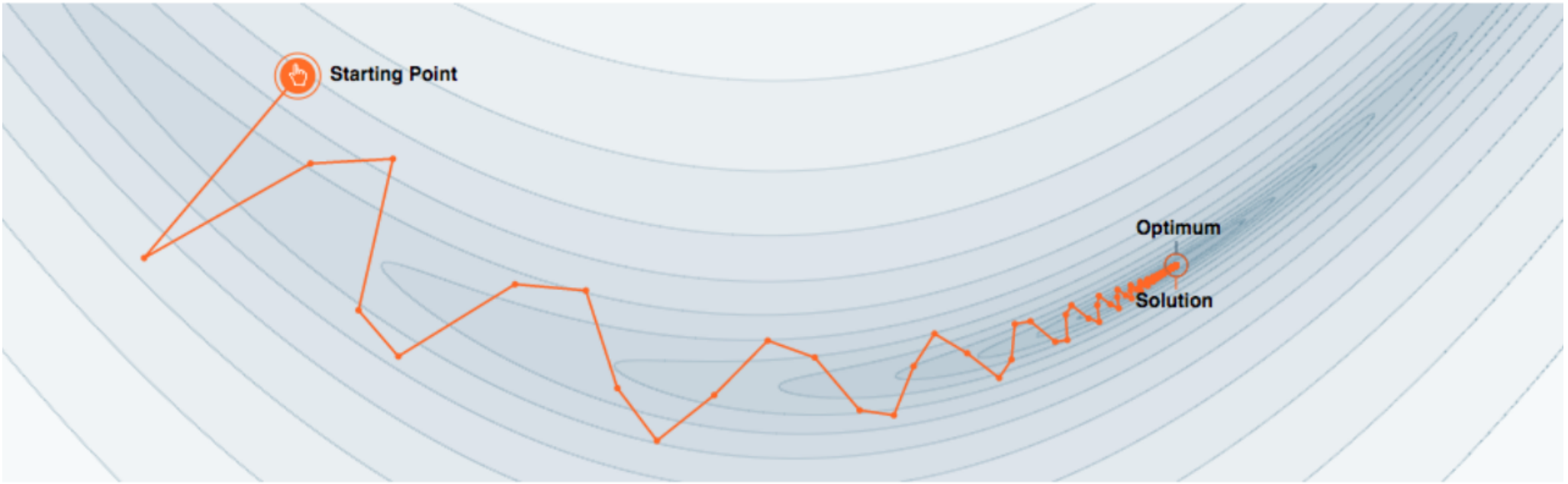


Figure: Optimization with momentum (Source: distill.pub)

Sebastian Ruder. Optimization for Deep learning, 2017

Adagrad

- Previous methods: **Same learning rate** η for all parameters θ.
- Adagrad [Duchi et al., 2011] **adapts** the learning rate to the parameters (**large** updates for **infrequent** parameters, **small** updates for **frequent** parameters).
- SGD update: $\theta_{t+1} = \theta_t - \eta \cdot g_t$
  - $g_t = \nabla_{\theta_t} J(\theta_t)$
- Adagrad divides the learning rate by the **square root of the sum of squares of historic gradients**.
- Adagrad update:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3}$$

  - $G_t \in \mathbb{R}^{d \times d}$: diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t
  - ϵ: smoothing term to avoid division by zero
  - $\odot$: element-wise multiplication

Sebastian Ruder. Optimization for Deep learning, 2017

Adagrad

- Pros
  - Well-suited for dealing with **sparse data**.
  - Significantly **improves robustness** of SGD.
  - Lesser need to manually tune learning rate.
- Cons
  - **Accumulates squared gradients** in denominator. Causes the learning rate to **shrink** and become **infinitesimally small**.

Sebastian Ruder. Optimization for Deep learning, 2017

RMSprop

- RMSprop update:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

  - γ: decay parameter; typically set to 0.9

Sebastian Ruder. Optimization for Deep learning, 2017

Adam

- Adaptive Moment Estimation (Adam) [Kingma and Ba, 2015] also stores **running average of past squared gradients** v_t like Adadelta and RMSprop.
- Like Momentum, stores **running average of past gradients** m_t.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{13}$$

  - m_t: first moment (mean) of gradients
  - v_t: second moment (uncentered variance) of gradients
  - β_1, β_2: decay rates

Sebastian Ruder. Optimization for Deep learning, 2017

Adam

- m_t and v_t are initialized as 0-vectors. For this reason, they are biased towards 0.
- Compute bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{14}$$

- Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{15}$$

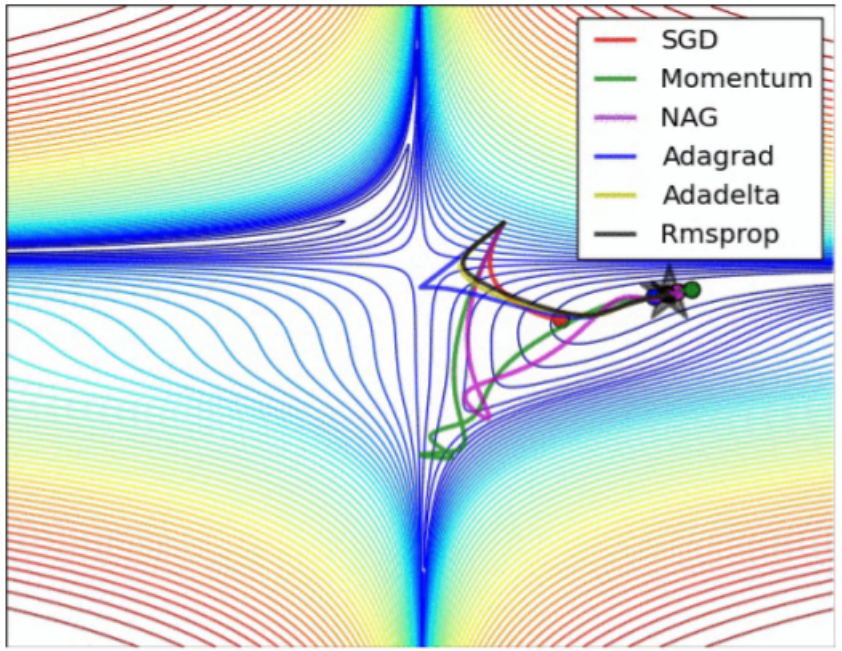
Sebastian Ruder. Optimization for Deep learning, 2017

Other optimizers

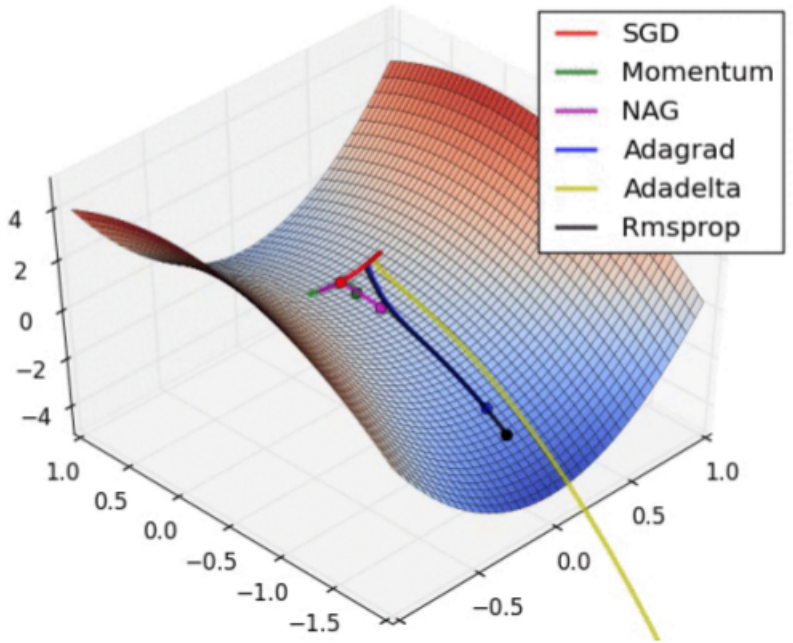
Method	Update equation
	$g_t = \nabla_{\theta_t} J(\theta_t)$
SGD	$\Delta\theta_t = -\eta \cdot g_t$
	$\theta_t = \theta_t + \Delta\theta_t$
Momentum	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta g_t$
NAG	$\Delta\theta_t = -\gamma\, v_{t-1} - \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
Adagrad	$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$
Adadelta	$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
RMSprop	$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$
Adam	$\Delta\theta_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$

Sebastian Ruder. Optimization for Deep learning, 2017

Other optimizers



(a) SGD optimization on loss surface contours



(b) SGD optimization on saddle point

Look at animation:
http://ruder.io/optimizing-gradient-descent/index.html#visualizationofalgorithms

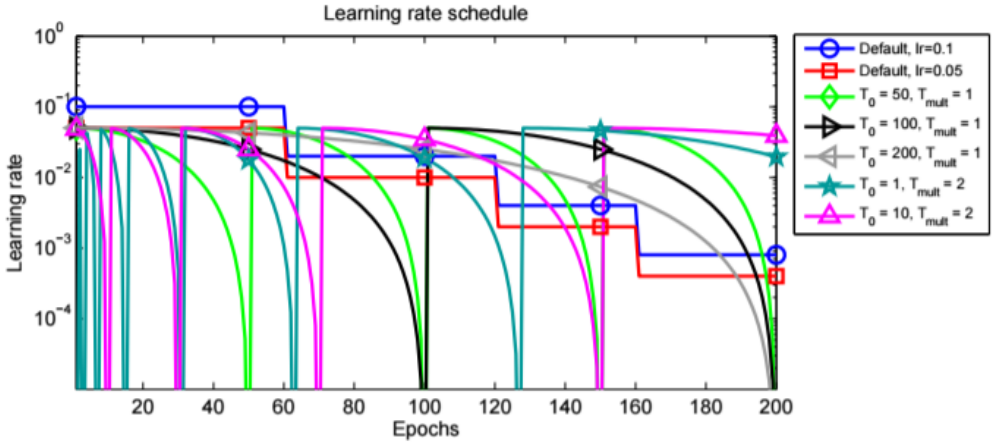
Sebastian Ruder. Optimization for Deep learning, 2017

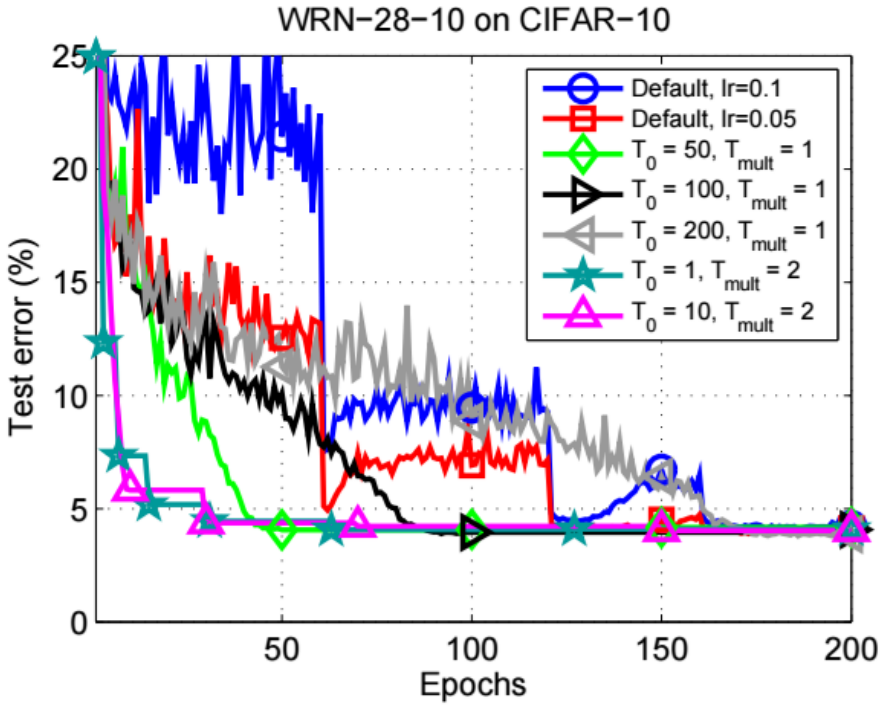
In practice

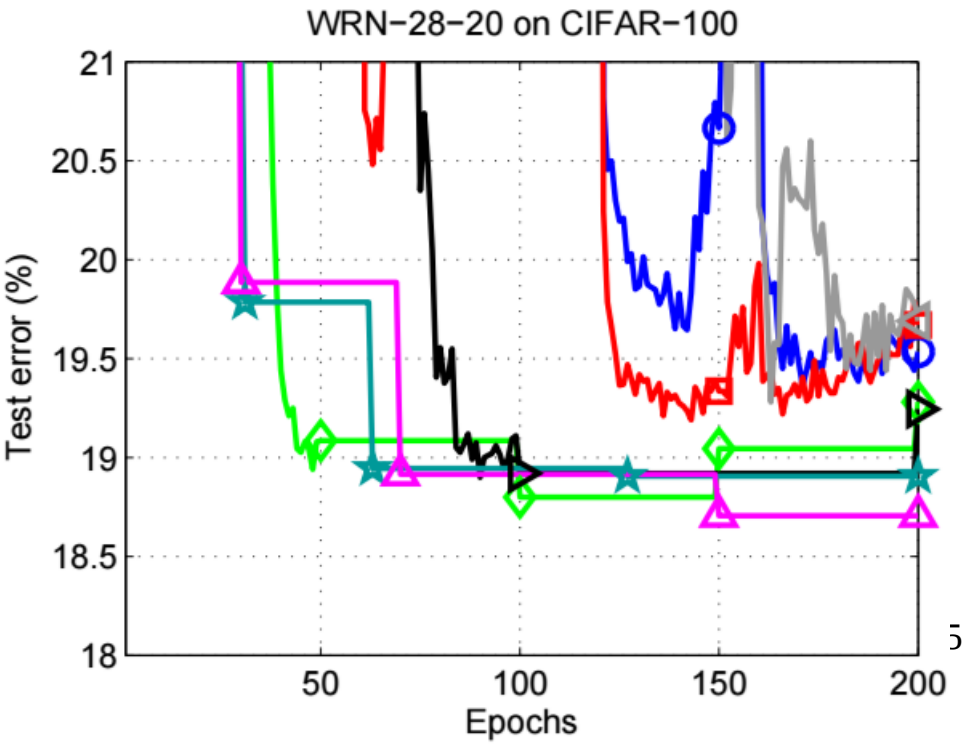
- Adaptive methods: better for sparse data (NLP)
- Adam – best default choice
  - Usually converges faster than SGD
  - Less sensitive to non-optimal lr / lr annealing than SGD
- For the best results – experiment:
  - SGD+Momentum with carefully tuned lr / lr annealing / momentum can be better [Zhang et al. 2017]
  - Adam still benefits from lr / lr annealing tuning
  - Adam with 2 restarts beats SGD+Momentum [Denkowski and Neubig, 2017]
    - train until convergence, halve the lr and restart
    - restart causes the optimizer to “forget” the per-parameter learning rates and start fresh

Cyclical Learning rate schedule

- Loshchilov, I., Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts

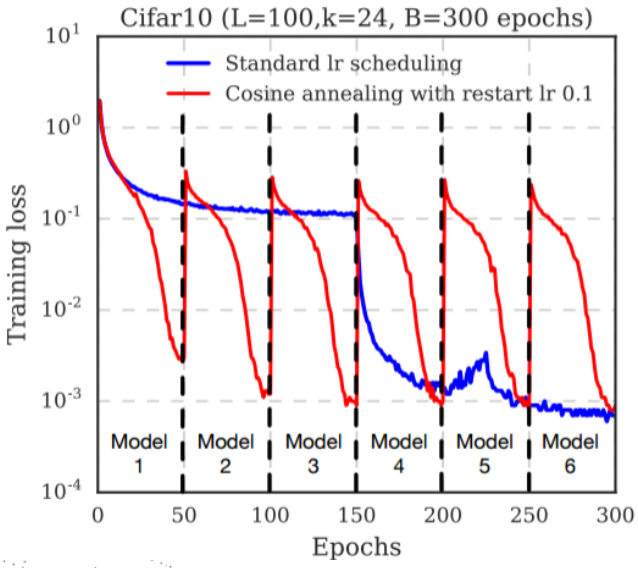


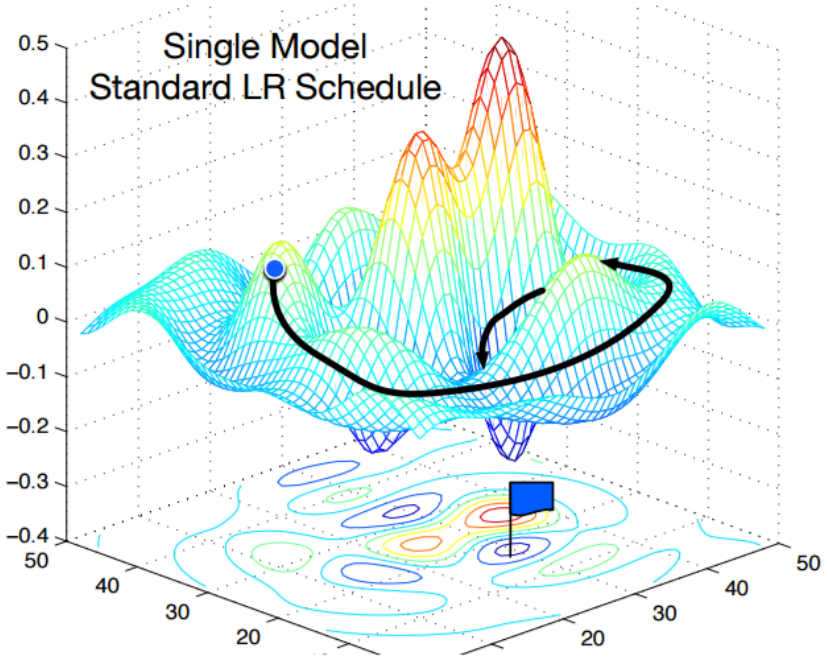


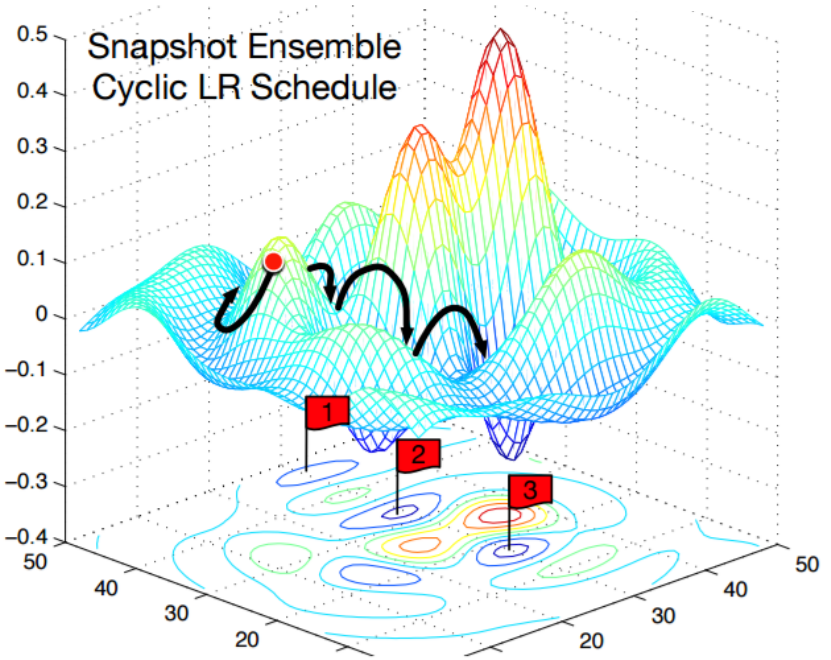


Snapshot Ensembles

- Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017). Snapshot Ensembles: Train 1, get M for free







Plan

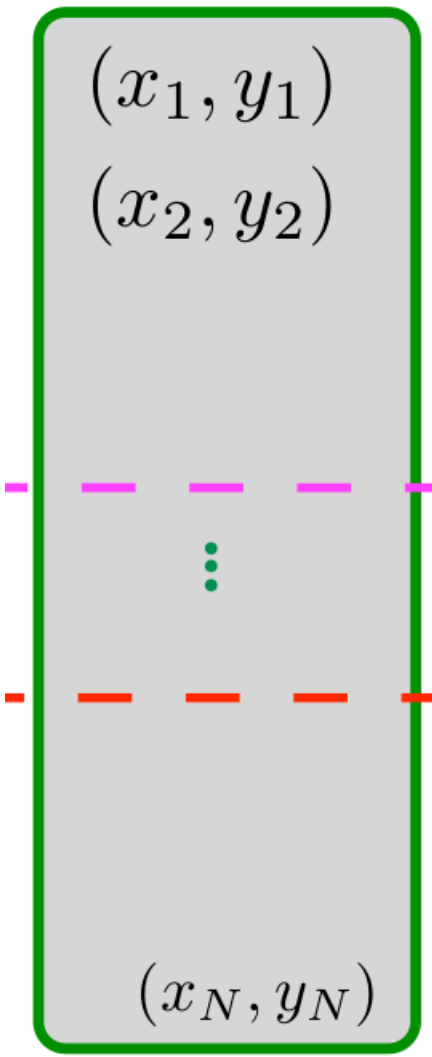
- Data normalization, weights initialization.
- Advanced optimization
- **Cross-validation**

Model selection how to

Make sure examples are in random order

Split data D in 3: D_{train} D_{valid} D_{test}

$D=$



Training set D_{train}

Validation set D_{valid}

Test set D_{test}

Model selection meta-algorithm:

For each considered model (ML algo) $\mathbf{A}$:

For each considered hyper-parameter config λ:

- train model $\mathbf{A}$ with hyperparams λ on D_{train}

$$\hat{f}_{\mathbf{A}_\lambda} = \mathbf{A}_\lambda(D_{\text{train}})$$

- evaluate resulting predictor on D_{valid} (with preferred evaluation metric)

$$e_{\mathbf{A}_\lambda} = \hat{R}(\hat{f}_{\mathbf{A}_\lambda}, D_{\text{valid}})$$

Locate $\mathbf{A}^*, \lambda^*$ that yielded best $e_{\mathbf{A}_\lambda}$

Either return $f^* = f_{\mathbf{A}^*_{\lambda^*}}$

Or retrain and return

$$f^* = \mathbf{A}^*_{\lambda^*}(D_{\mathbf{train}} \cup D_{\mathbf{valid}})$$

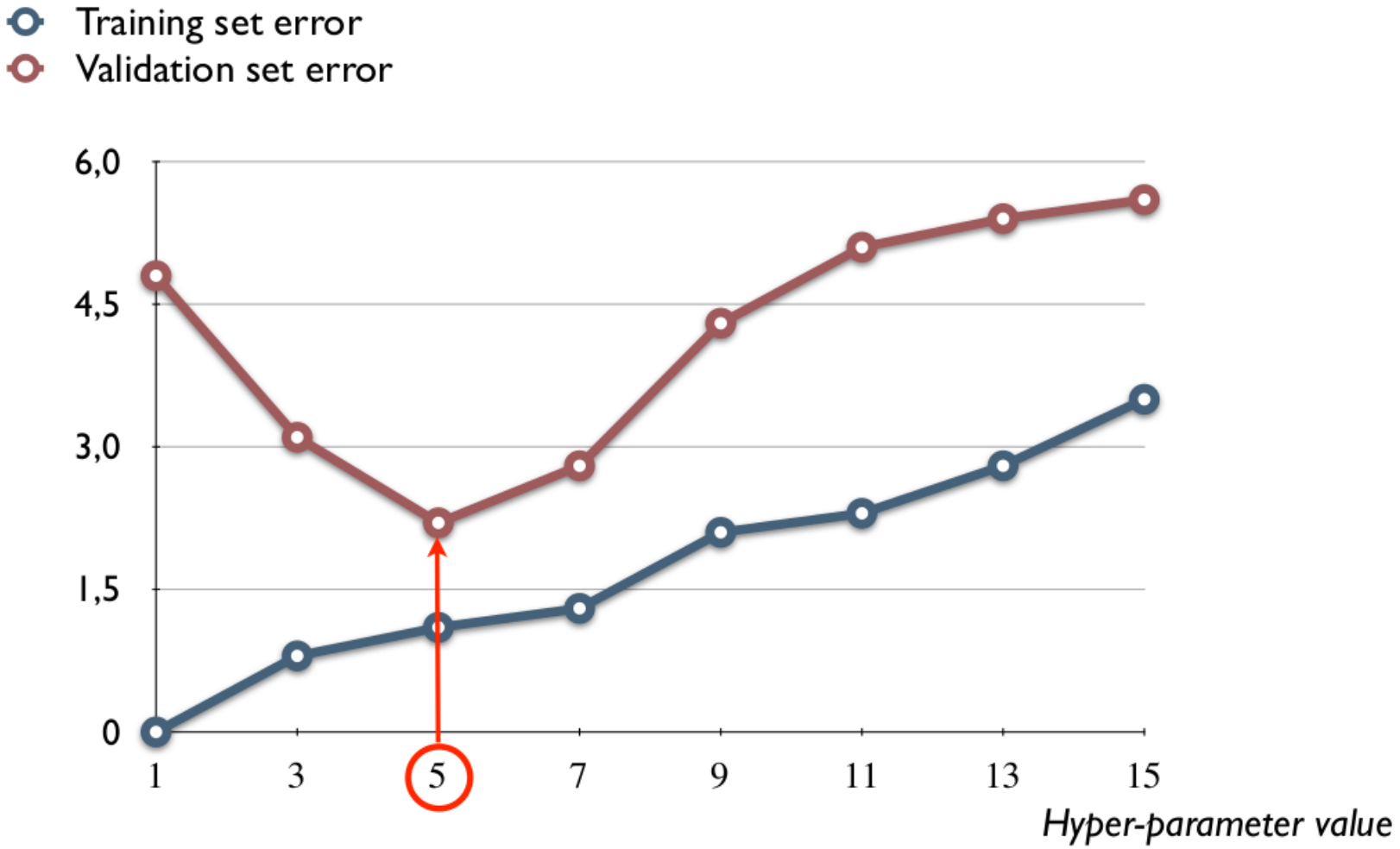
Finally: compute unbiased estimate of generalization performance of f^* using D_{test}

$$\hat{R}(f^*, D_{\mathbf{test}})$$

D_{test} must never have been used during training or model selection to select, learn, or tune anything.

Pascal Vincent, Introduction to Machine Learning. Deep Learning Summer School, Montreal 2015

Hyperparameter selection

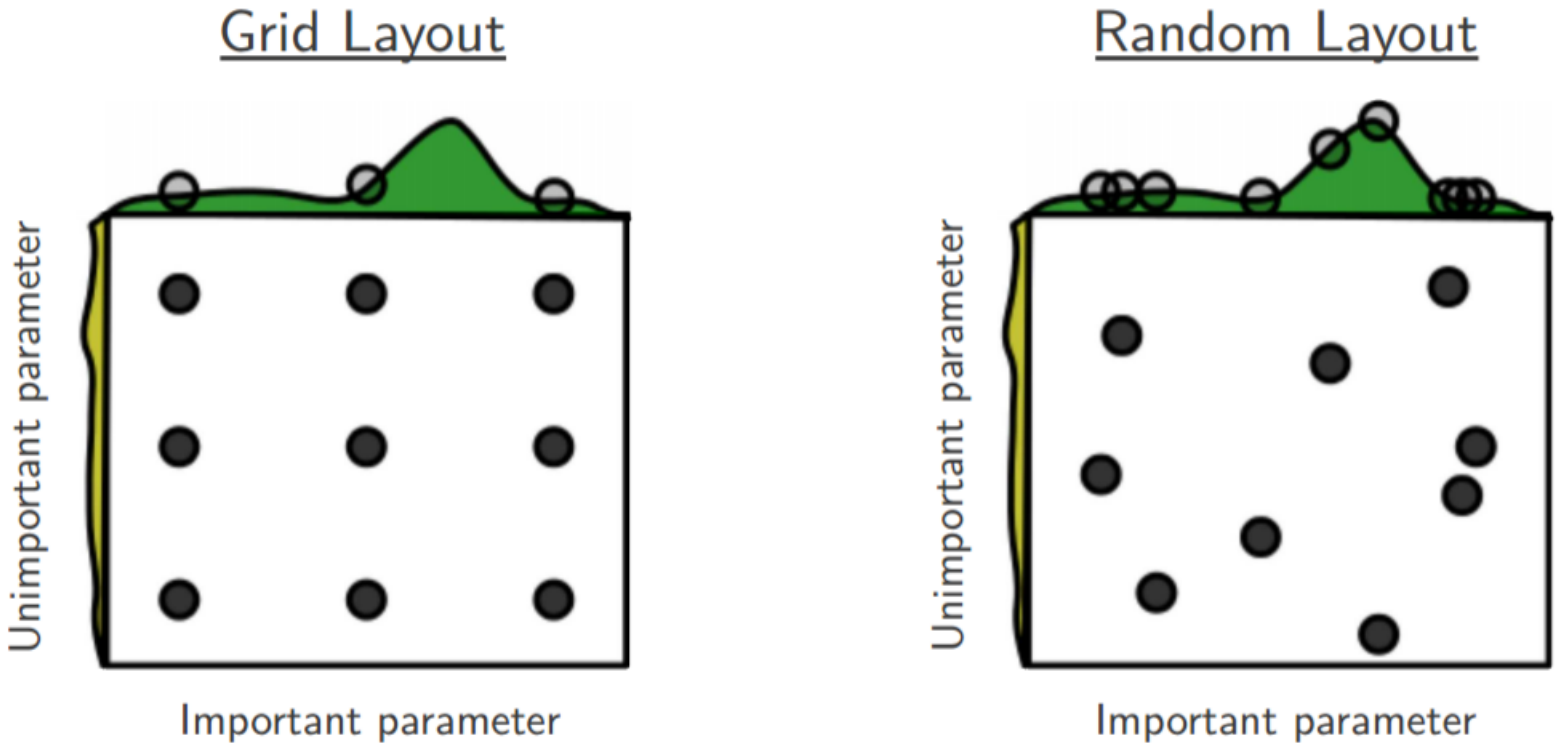


Hyper-parameter value which yields smallest error on validaiton set is 5
(it was 1 for the training set)

Pascal Vincent, Introduction to Machine Learning. Deep Learning Summer School, Montreal 2015

Use random search

- Usually only some hyperparameters are important
  - Want to try many values for them



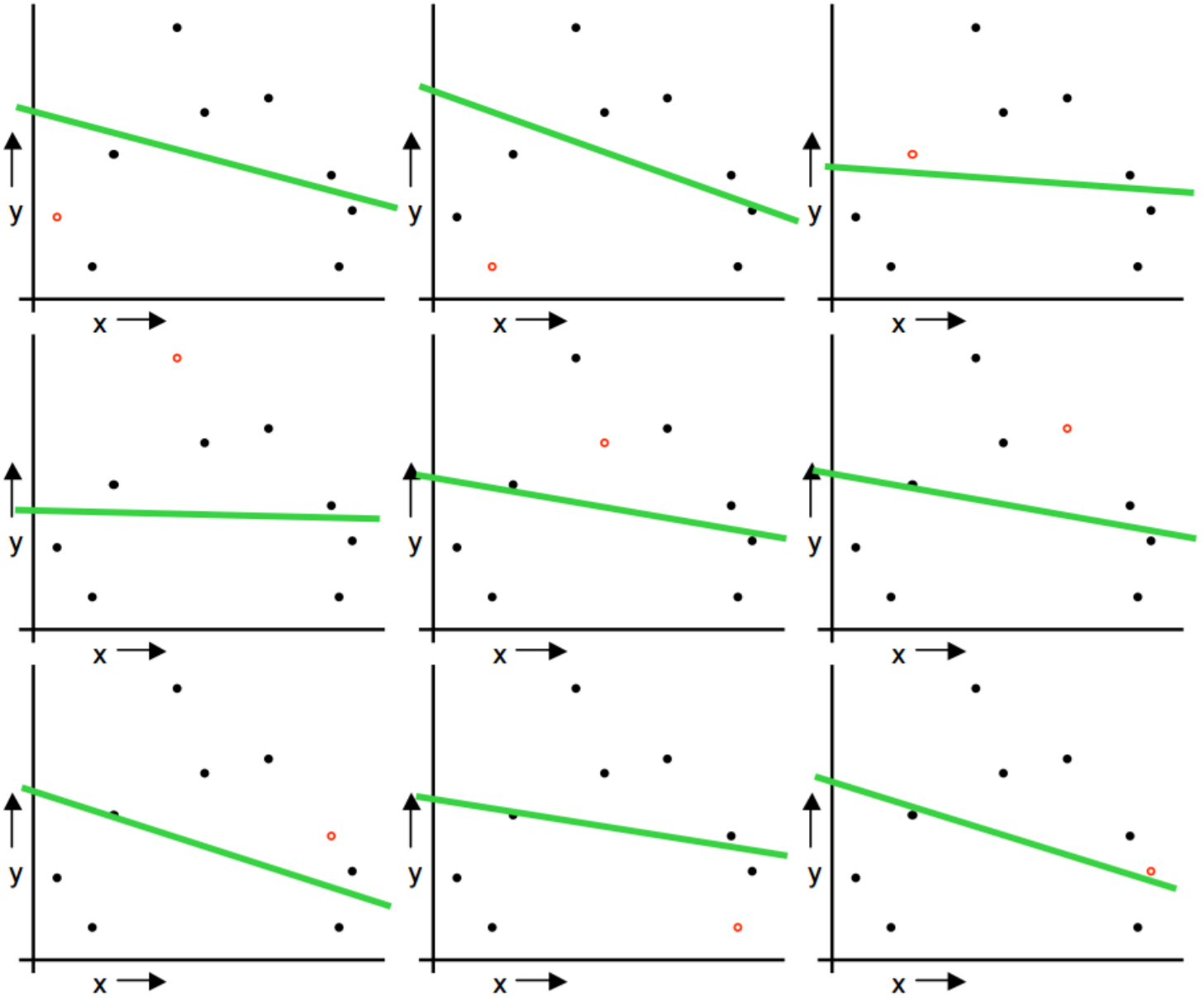
Bergstra and Bengio. Random Search for Hyper-Parameter Optimization, 2012.

Train/dev/test split

- Pros:
  - Simple
  - Fast
- Cons:
  - If dev/test sets are small, the expected risk estimates on them are unreliable (consider 1 ex)
  - If dev/test sets are large (say 25% / 25%), train set becomes smaller (50%) and we select optimal model for this train size and use it for larger train size (train+dev) => wastes data
  - Test set is lost for training [otherwise cannot estimate generalization error/expected risk!] => wastes data

Balancing train/dev/test sets is extremely problemmatic if dataset is not very large!

Leave-one-out Cross Validation



For k=1 to R

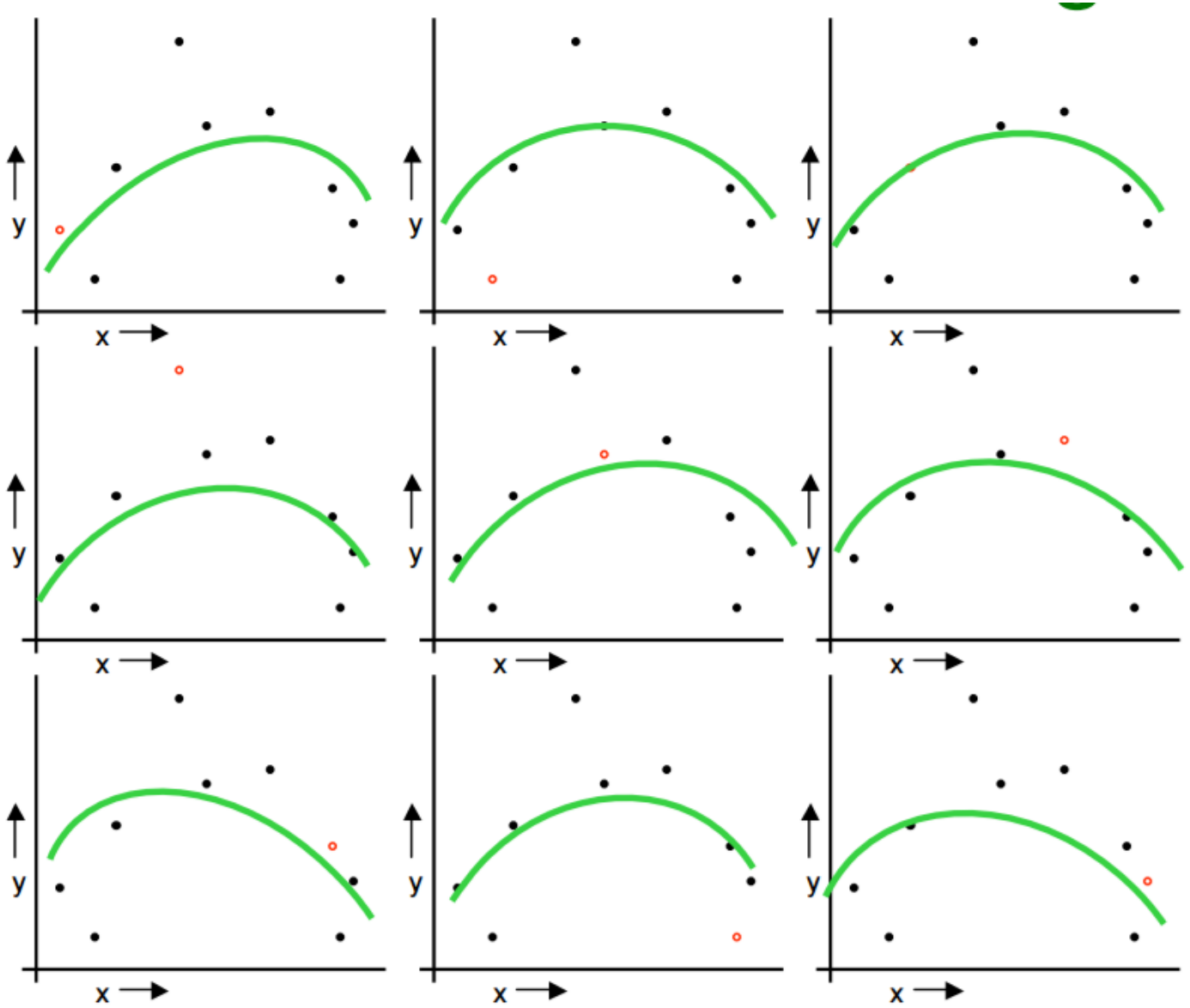
1. Let (x_k, y_k) be the k^{th} record
2. Temporarily remove (x_k, y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k, y_k)

When you've done all points, report the mean error.

$$MSE_{LOOCV} = 2.12$$

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

Leave-one-out Cross Validation



For k=1 to R

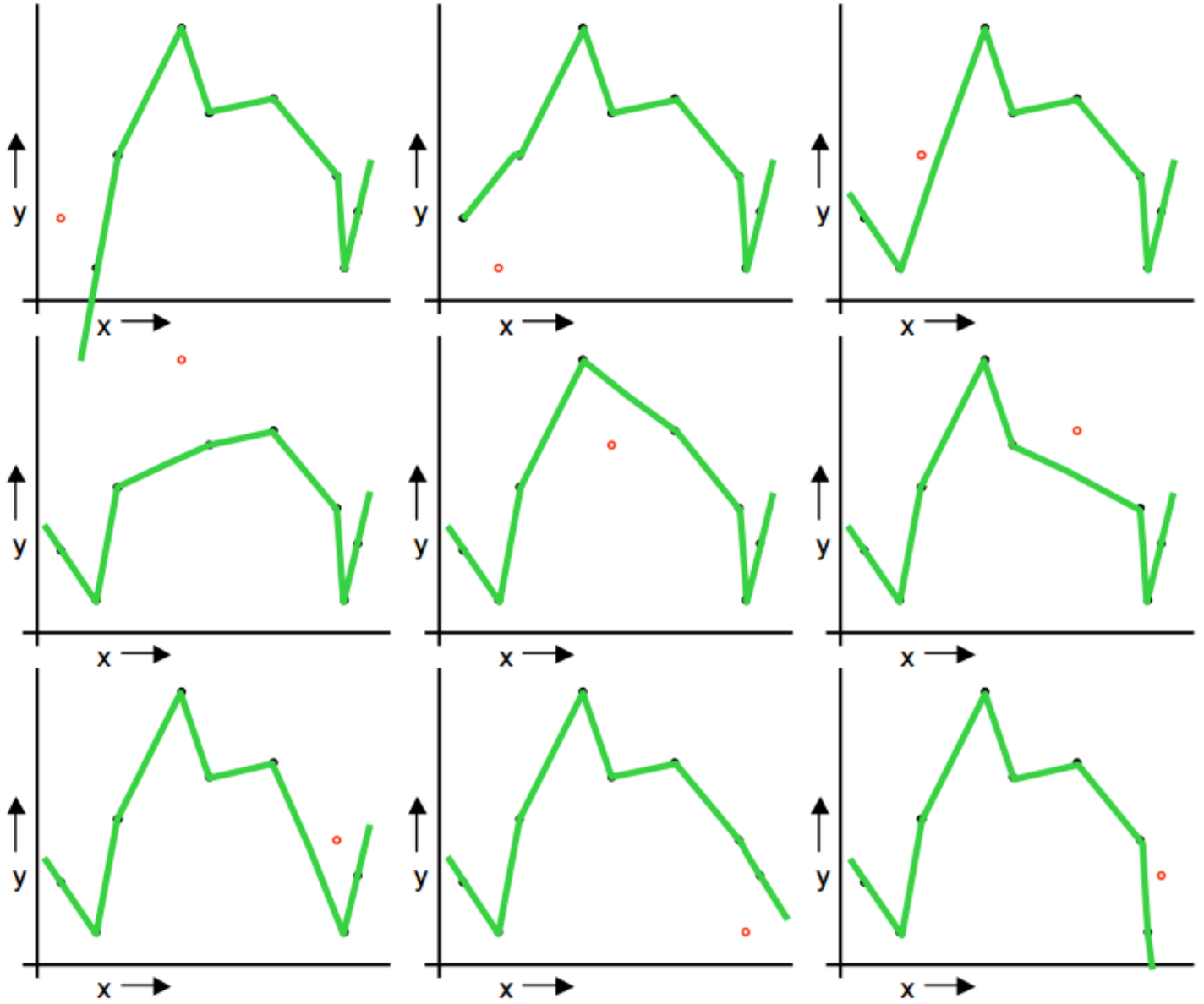
1. Let (x_k, y_k) be the k^{th} record
2. Temporarily remove (x_k, y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k, y_k)

When you've done all points, report the mean error.

$MSE_{LOOCV} = 0.962$

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

Leave-one-out Cross Validation



For k=1 to R

1. Let (x_k, y_k) be the k^{th} record
2. Temporarily remove (x_k, y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k, y_k)

When you've done all points, report the mean error.

$MSE_{LOOCV} = 3.33$

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

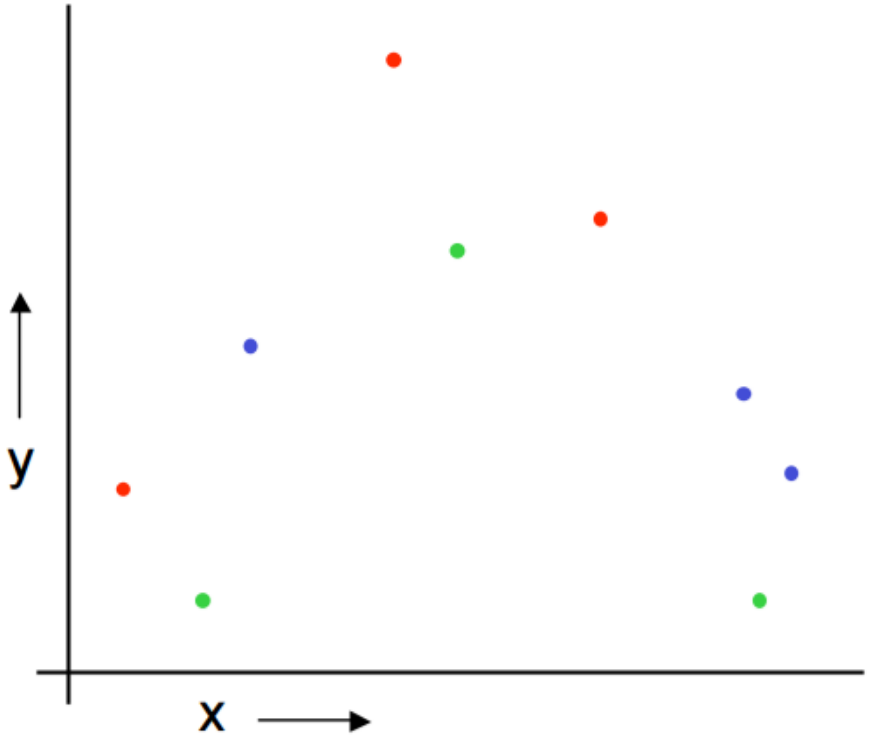
LOOCV

- Pros:
  - Doesn’t waste data [finally can train on whole dataset, expected risk is already measured during CV!]
- Cons:
  - Very slow [training N models]
  - Strange results sometimes:
    - Balanced dataset with 2 classes, most frequent class classifier 0% acc [should be 50%]
      - Train: {n/2 – 1 of A, n/2 of B} Test: {1 of A, 0 of B}
    - Duplicate dataset, 1-NN classifier => 100% acc !?

K-fold Cross Validataion

k-fold Cross Validation

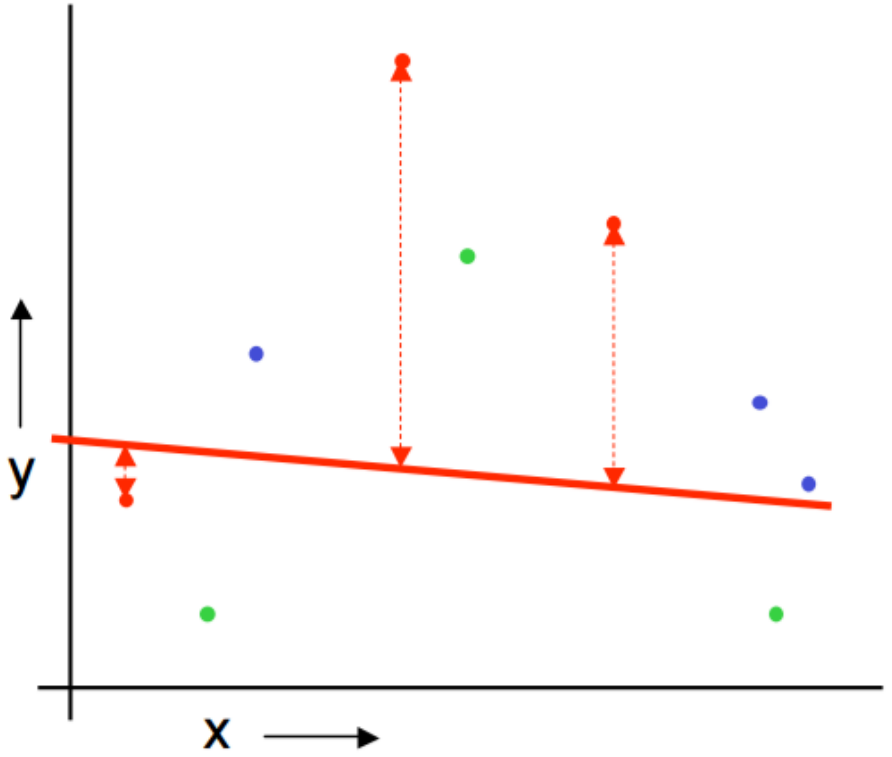
Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)



Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

K-fold Cross Validataion

k-fold Cross Validation



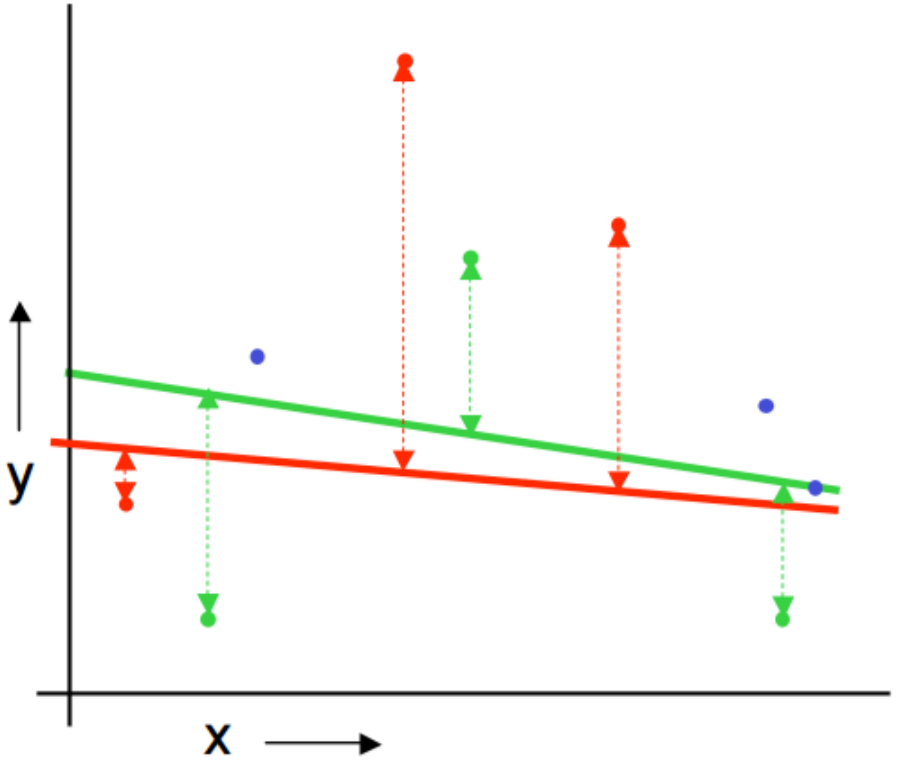
Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

K-fold Cross Validataion

k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

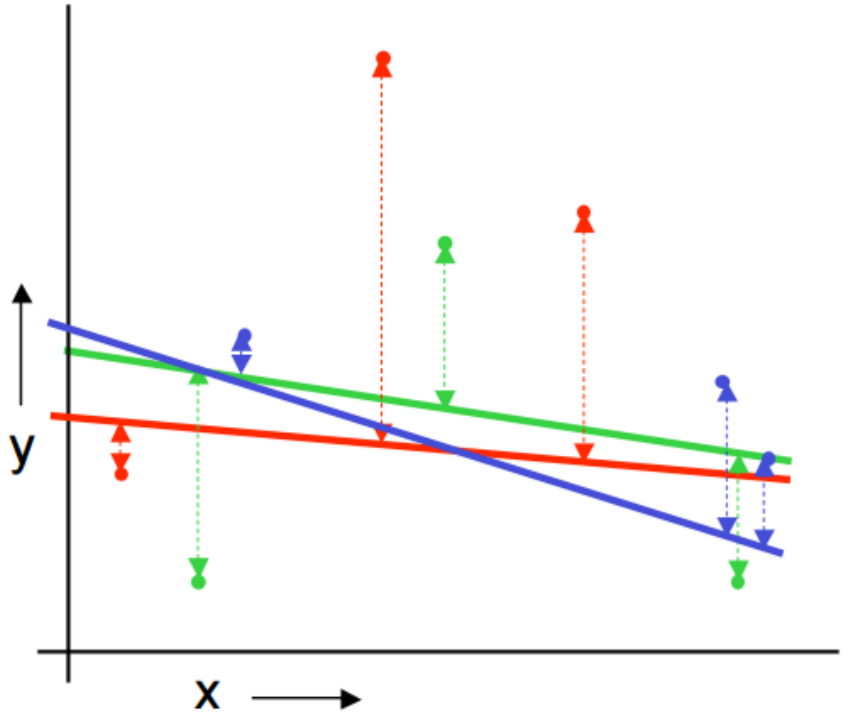
For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

K-fold Cross Validataion

k-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

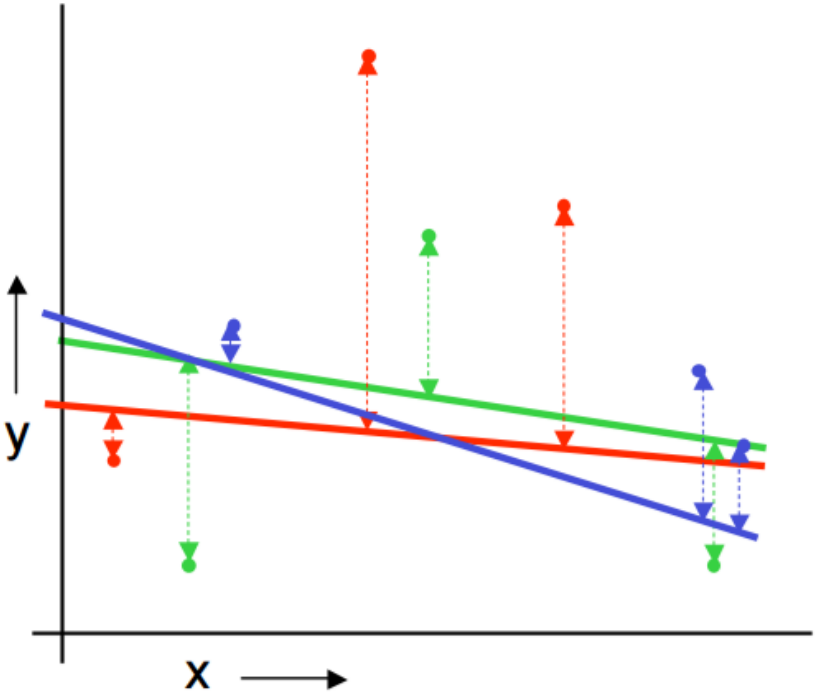
For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

K-fold Cross Validataion

k-fold Cross Validation



Linear Regression

$MSE_{3FOLD}=2.05$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

Andrew W. Moore. Cross-validation for detecting and preventing overfitting.

10-fold Cross Validataion

10-fold CV is usually used

- Pros:
  - vs. train/dev/test: wastes only 10% of the data
  - vs. LOOCV: only 10 times slower than train/dev/test (compared to N times slower for LOOCv)
- Cons:
  - vs. train/dev/test: 10 times slower
  - vs. LOOCV: wastes 10% of the data