14.332.435/16.332.530
Introduction to Deep Learning

Lecture 3
Machine Learning Basics (2)

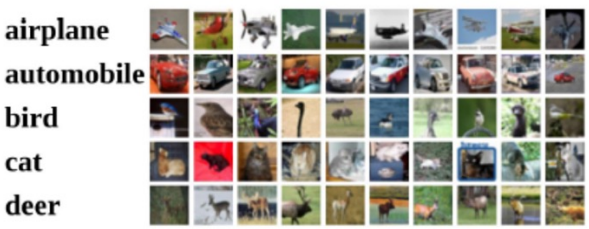
Yuqian Zhang

Department of Electrical and Computer Engineering

Recall Last Lecture – Linear Classifier

Update W

Training Examples



ML Model ($f(x_{train}, W)$)

Array of 32x32x3 numbers
(3072 numbers total)

Loss Function

W: Model weights (Parameters)

$$\underbrace{f(x,W)}_{10\times 1} = \underbrace{W}_{10\times 3072}\overbrace{x}^{3072\times 1} + \underbrace{b}_{10\times 1}$$

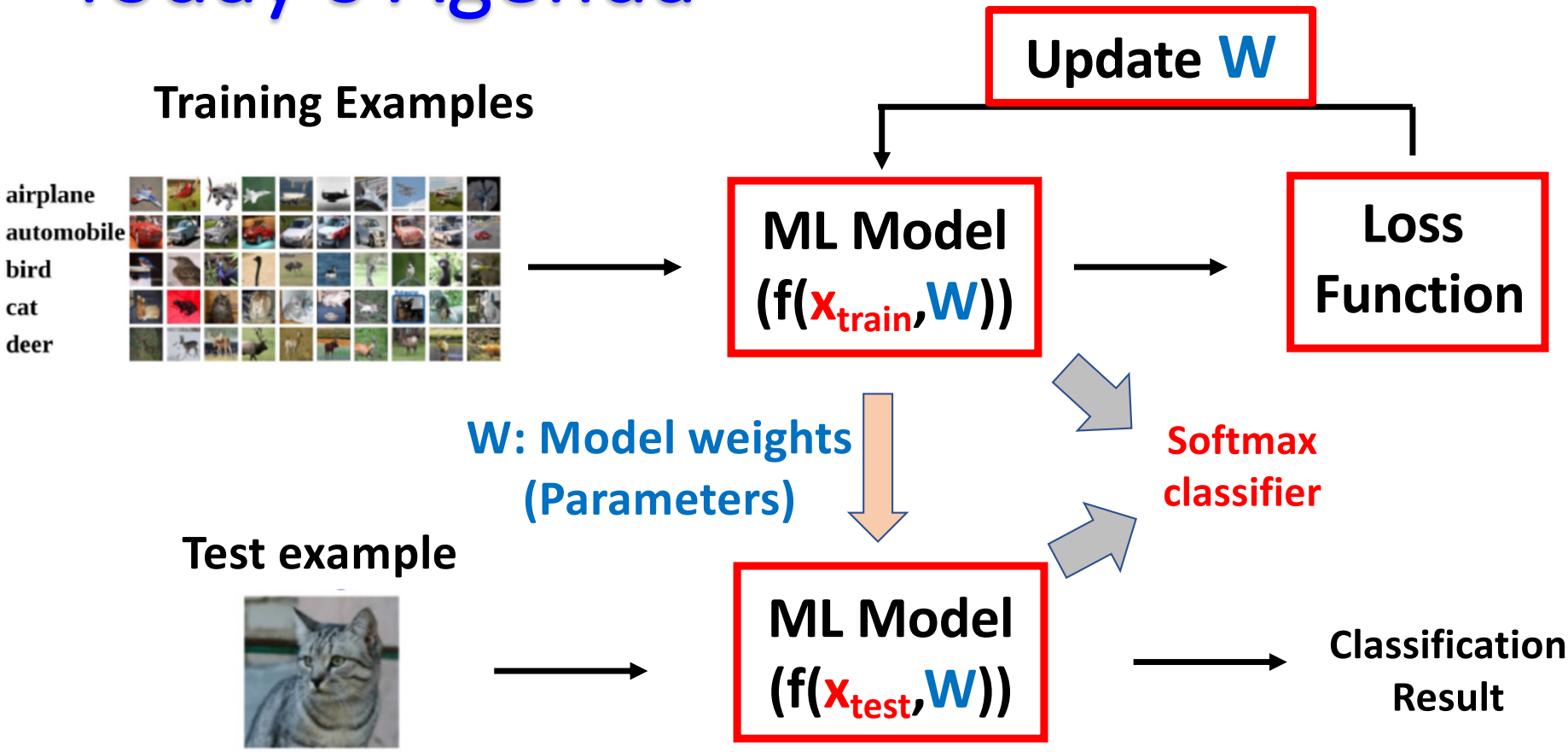
Test example

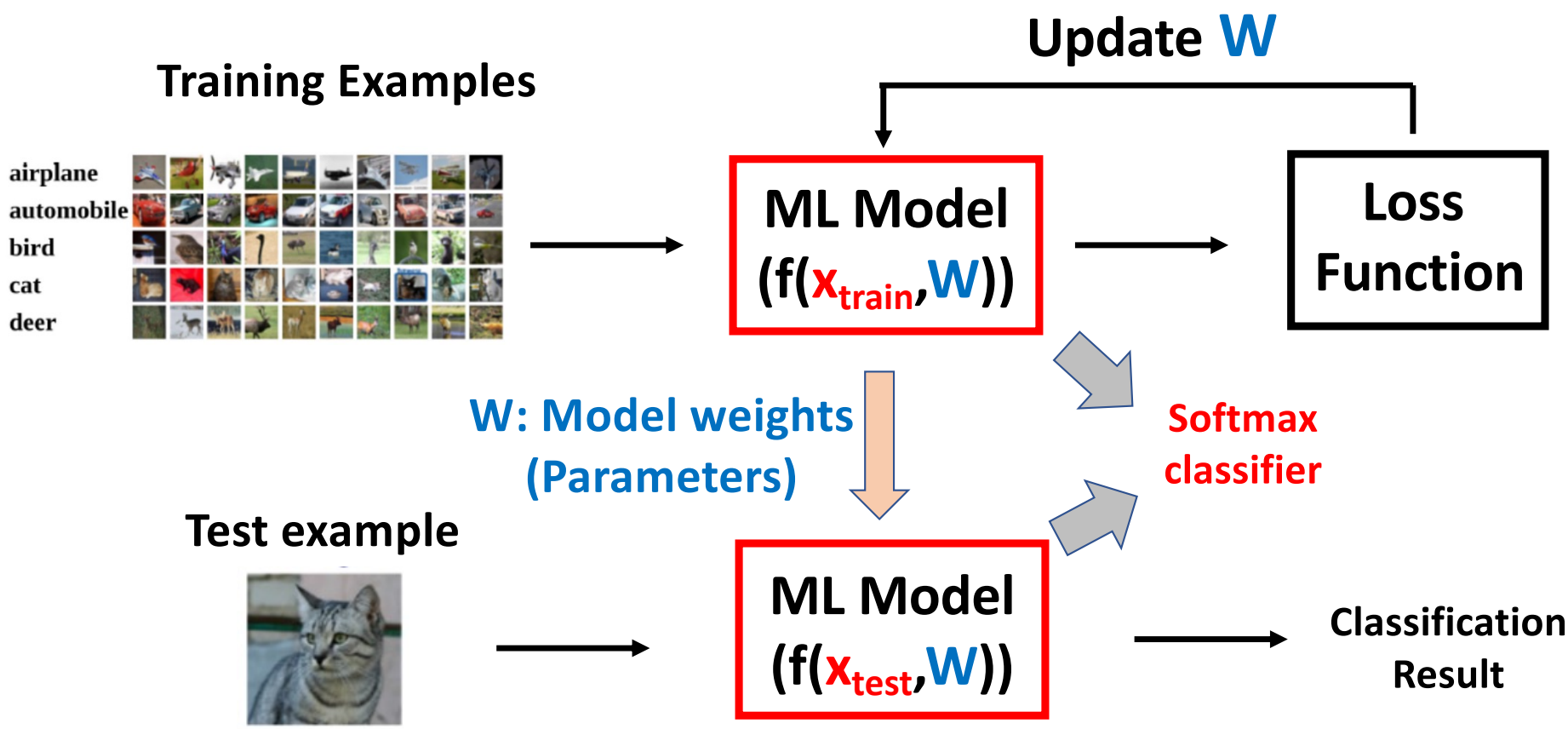
32x32x3 (RGB channel)
Flatten to length-3072 vector **x**

ML Model ($f(x_{test}, W)$)

Classification Result

Today's Agenda




Update W
Training Examples
airplane
automobile
bird
cat
deer
ML Model
(f(x_train,W))
Loss
Function
W: Model weights
(Parameters)
Softmax
classifier
Test example
ML Model
(f(x_test,W))
Classification
Result

Softmax Classifier

$$P(Y = k|X = \mathbf{x}_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax function

- Build upon linear classifier
- Map score of class *k* to probability of being in this class
- The probabilities of being in different classes sum up to 1

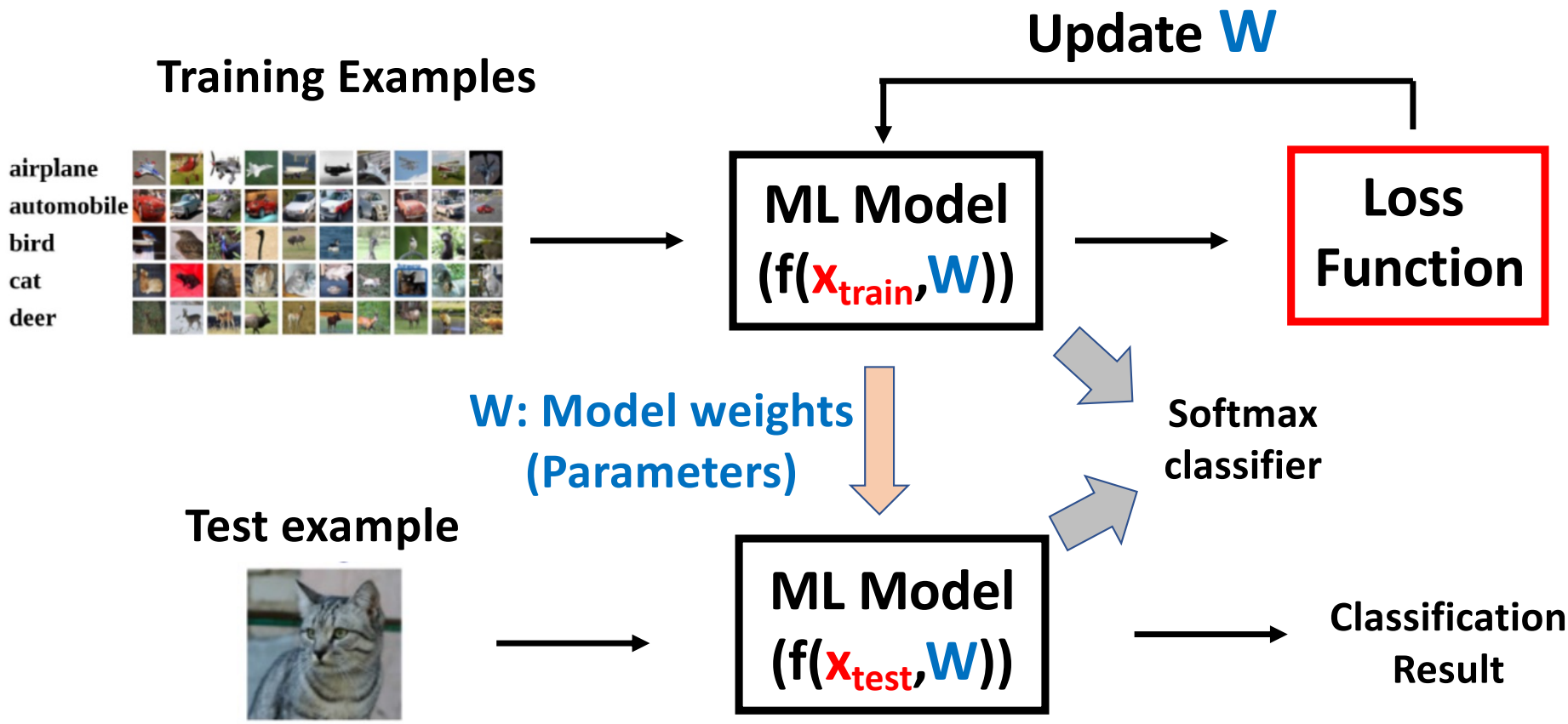
	Scores s_i		Unnormalized probabilities (Pr must be >0)		Probabilities (Total of Pr must be 1)
cat	3.2	exp →	24.5	normalize →	0.13
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

Softmax in Practice

- Numeric stability must be considered for exp terms

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}} \qquad \log C = -\max_j f_j$$

```
f = np.array([123, 456, 789]) # example with 3 classes and each having large s
p = np.exp(f) / np.sum(np.exp(f)) # Bad: Numeric problem, potential blowup

# instead: first shift the values of f so that the highest number is 0:
f -= np.max(f) # f becomes [-666, -333, 0]
p = np.exp(f) / np.sum(np.exp(f)) # safe to do, gives the correct answer
```


Update W
Training Examples
airplane
automobile
bird
cat
deer
ML Model
(f(x_train,W))
Loss
Function
W: Model weights
(Parameters)
Softmax
classifier
Test example
ML Model
(f(x_test,W))
Classification
Result

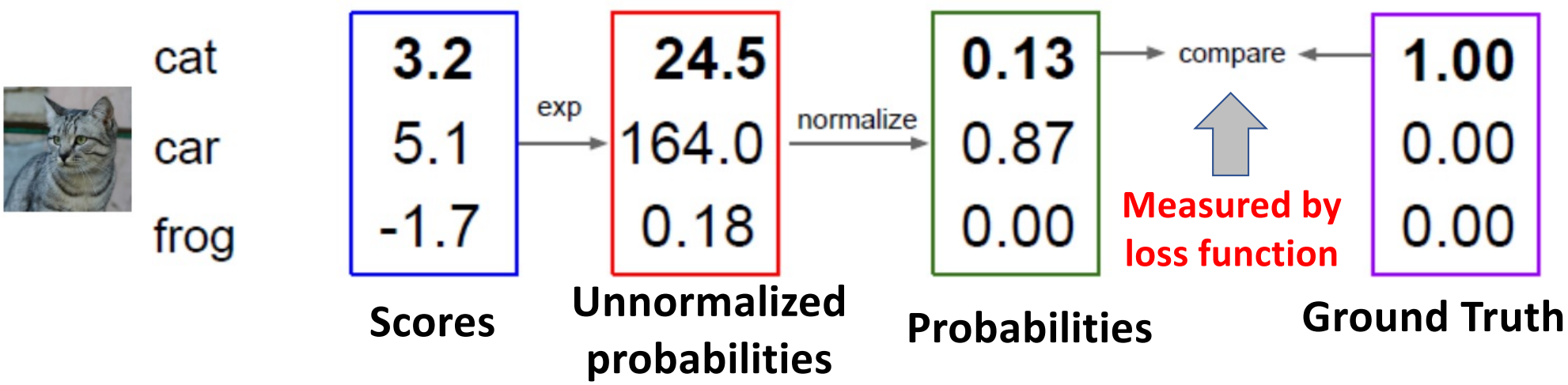
Loss function

$$L = \frac{1}{N} \sum L_i(\boxed{f(\mathbf{x_i}, \mathbf{W})}, y_\mathbf{i})$$

$\hat{y}_i$ Model para.

# of examples — *i*-th example — *Labe of i*-th example

- Tells how good current classifier is
- Loss over the dataset is the average loss for all examples



Mean Absolution Error (MAE) Loss

- Also called L1 loss

$$L_i = |\hat{y}_i - y_i|$$

```
def L1(yHat, y):
    return np.sum(np.absolute(yHat - y))
```

Mean Square Error (MSE) Loss

- Also called L2 loss

$$L_i = (\hat{y}_i - y_i)^2$$

```
def MSE(yHat, y):
    return np.sum((yHat - y)**2) / y.size
```

Cross Entropy Loss

- Negative log likelihood of the correct class as the loss

$$L_m = -logP(Y = y_i | X = \mathbf{x}_i) = -log\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}$$

where y_i is the correct label

$$L = -\frac{1}{N}\left[\sum_{i=1}^{N}\sum_{k=1}^{K}\boxed{\mathbf{1}\{\boldsymbol{y_i} = \boldsymbol{k}\}}log\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right]$$

Indicator function **1**{x}: **1**{x=true}=1

1{x=false}=0

Example

- For a single training example, the true label is *one-hot* as [1 0 0 0 0], while the prediction of softmax classifier is [0.1 0.5 0.1 0.1 0.2]. Calculate cross-entropy loss

Cross-entropy Loss=

-(1xlog(0.1)+0xlog(0.5)+0xlog(0.1)+0xlog(0.1)+0xlog(0.2))

=-log(0.1)

=2.303

Derivation

- In information theory, the cross-entropy between a true distributions *p* and an estimated distribution *q* is calculated as

$$H(p,q) = -\sum_{\forall x} p(x) log(q(x))$$

- In softmax, *p* is the ground truth one-hot label (0,0,..1,0,…0) and *q* is the prediction output vector f softmax

Another Example

- For a single training example, the true label is *one-hot* as [1 0 0 0 0], while the prediction of softmax classifier is [0.1 0.6 0.1 0.1 0.1]. Calculate cross-entropy loss

Cross-entropy Loss=

-(1xlog(0.1)+0xlog(0.6)+0xlog(0.1)+0xlog(0.1)+0xlog(0.2))

=-log(0.1)

=2.303

Q: What can you see from these two examples?

Regularization

- It is likely different W has the same loss
- Regularization helps to express preference
- Regularization helps to avoid overfitting

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(\mathbf{x_i}, \mathbf{W}), y_i) + \lambda R(\mathbf{W})$$

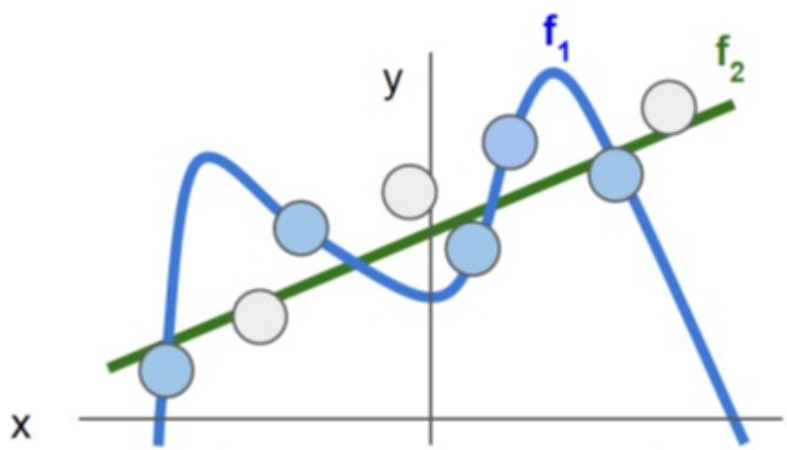
λ : Regularization strength

Data loss: Model prediction should match training data

Regularization: Prevent model from doing too well on training data

Avoid Overfitting

- Overfitting: Model tries to fit not only the regular relation between inputs and outputs, but also the sampling errors.
- Weight regularization helps to select simple models



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

Express Preference

$$\mathbf{x} = [1,1,1,1]$$

$$\mathbf{w}_1^{\mathrm{T}}\mathbf{x} = \mathbf{w}_2^{\mathrm{T}}\mathbf{x} = 1$$

$$\boldsymbol{R}(\boldsymbol{W}) = \sum \boldsymbol{W}_i^2$$

$$\mathbf{w}_1 = [1,0,0,0]$$

$$\boxed{\mathbf{w}_2 = [0.25,0.25,0.25,0.25]}$$

L2 regularization like to “spread out” the weights

Express Preference

- Achieve sparsity via L1 or SSL regularization

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda \cdot R(\boldsymbol{W}) + \lambda_g \cdot \sum_{l=1}^{L} R_g\left(\boldsymbol{W}^{(l)}\right). \quad (1)$$

Here $\boldsymbol{W}$ represents the collection of all weights in the DNN; $E_D(\boldsymbol{W})$ is the loss on data; $R(\cdot)$ is non-structured regularization applying on every weight, *e.g.*, ℓ_2-norm; and $R_g(\cdot)$ is the structured sparsity regularization on each layer. Because *group Lasso* can effectively zero out all weights in

Table 4: Sparsity and speedup of *AlexNet* on ILSVRC 2012

#	Method	Top1 err.	Statistics	conv1	conv2	conv3	conv4	conv5
1	ℓ_1	44.67%	sparsity	67.6%	92.4%	97.2%	96.6%	94.3%
			CPU ×	0.80	2.91	4.84	3.83	2.76
			GPU ×	0.25	0.52	1.38	1.04	1.36
2	SSL	44.66%	column sparsity	0.0%	63.2%	76.9%	84.7%	80.7%
			row sparsity	9.4%	12.9%	40.6%	46.9%	0.0%
			CPU ×	1.05	3.37	6.27	9.73	4.93
			GPU ×	1.00	2.37	4.94	4.03	3.05

Wen et. al NeurIPS16

Common Regularization

- Simple examples

  L2 regularization: $\boldsymbol{R}(\boldsymbol{W}) = \sum \boldsymbol{W}_{\boldsymbol{i}}^{\boldsymbol{2}}$

  L1 regularization: $\boldsymbol{R}(\boldsymbol{W}) = \sum |\boldsymbol{W}_{\boldsymbol{i}}|$

  Elastic net: L1+L2

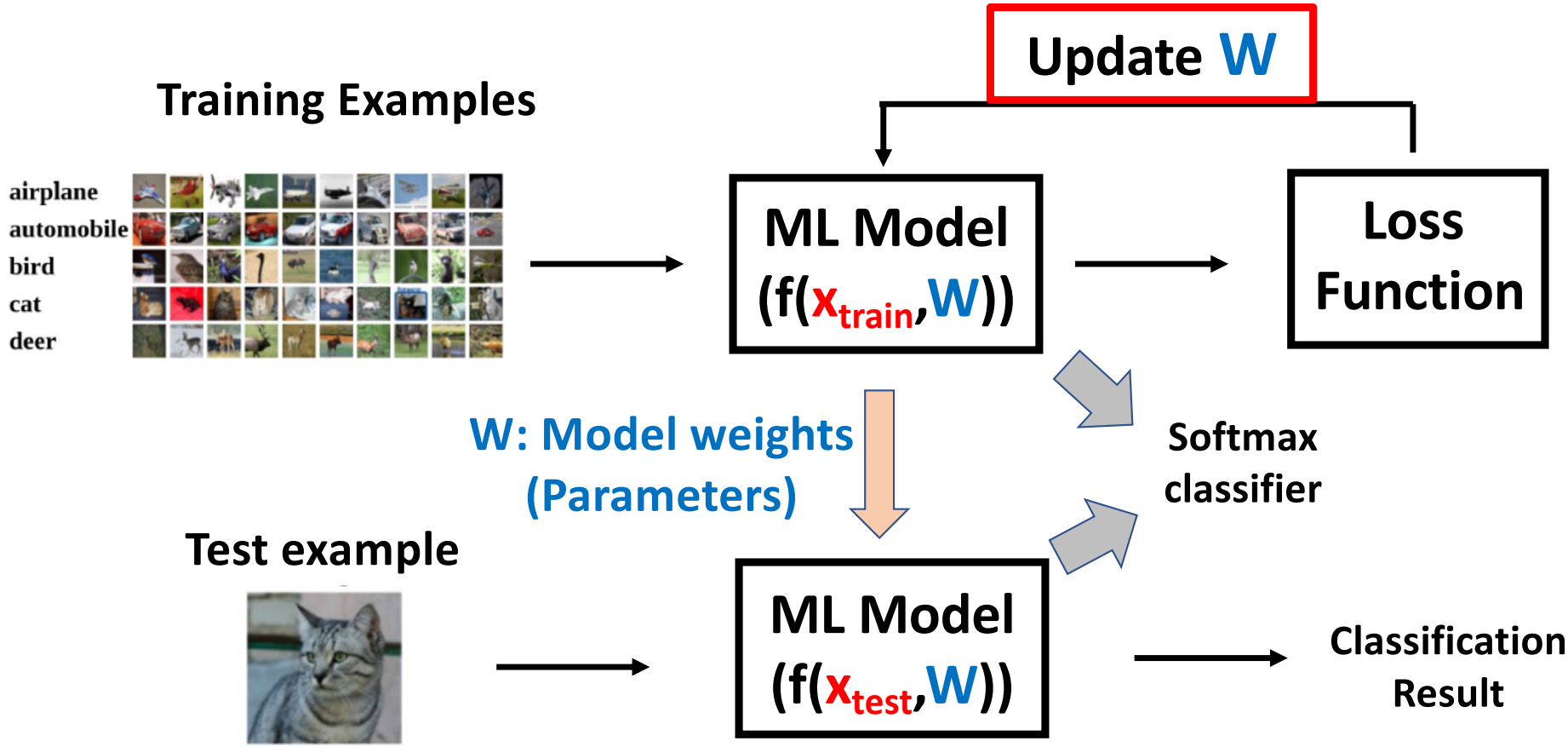
- More complex

  Dropout

  Batch normalization

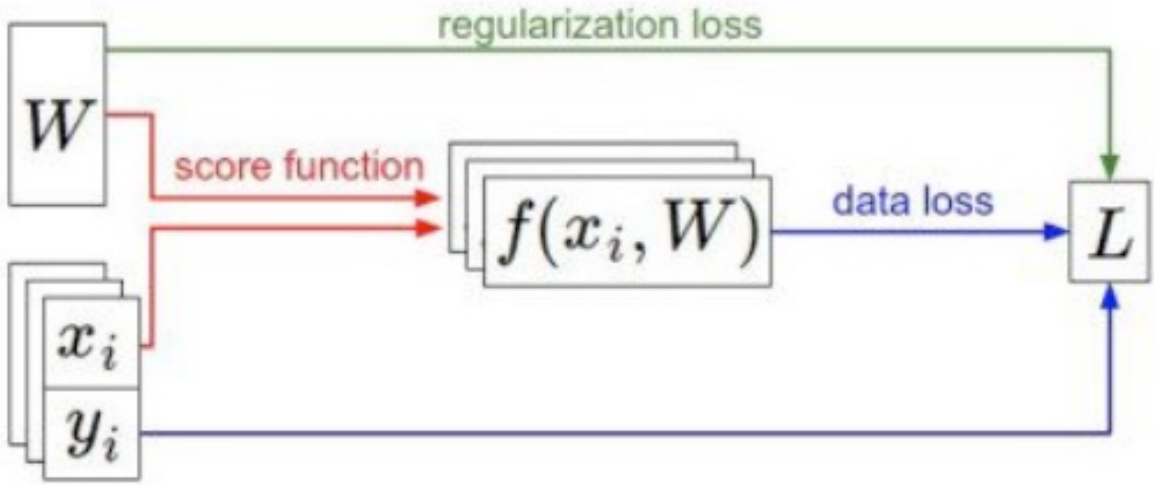
  Stochastic depth etc.

Will revisit in the lectures of DNNs

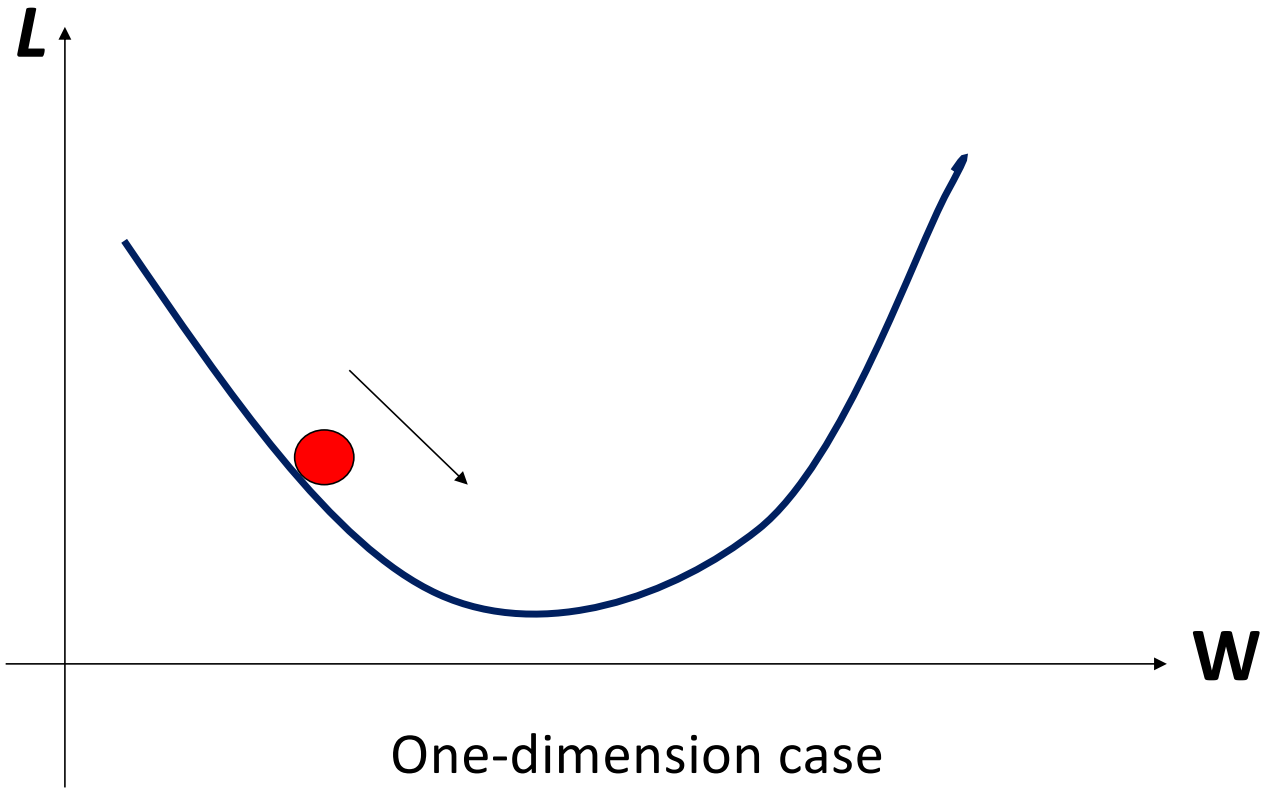

Update W
Training Examples
airplane
automobile
bird
cat
deer
ML Model
(f(x_train,W))
Loss
Function
W: Model weights
(Parameters)
Softmax
classifier
Test example
ML Model
(f(x_test,W))
Classification
Result

How to Find Best W

- The loss function lets us quantify the quality of any particular set of weights **W**. The goal of optimization is to find **W** that minimizes the loss function.



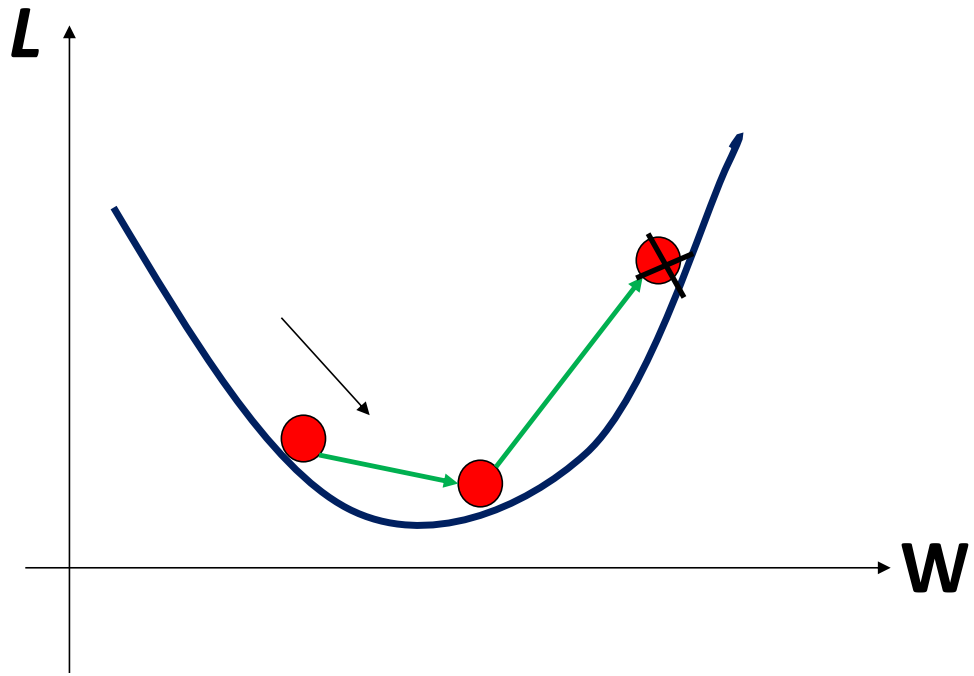
How to Find Proper W to Minimize L



One-dimension case

Strategy #1: Random Search (very bad)

- Simply try out many different random weights and keep track of what works best.



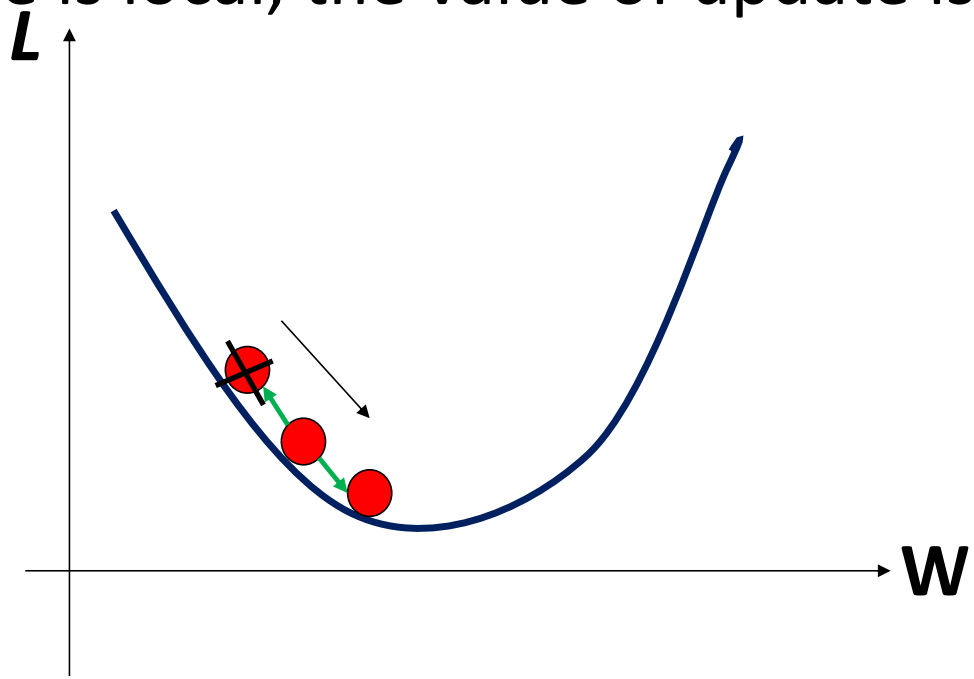
```python
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
  W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
  loss = L(X_train, Y_train, W) # get the loss over the entire training set
  if loss < bestloss: # keep track of the best solution
    bestloss = loss
    bestW = W
  print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

Random search

Update weight if loss decrease

Strategy #2: Random Local Search (still bad)

- The starting position is random
- The update is local, the value of update is random



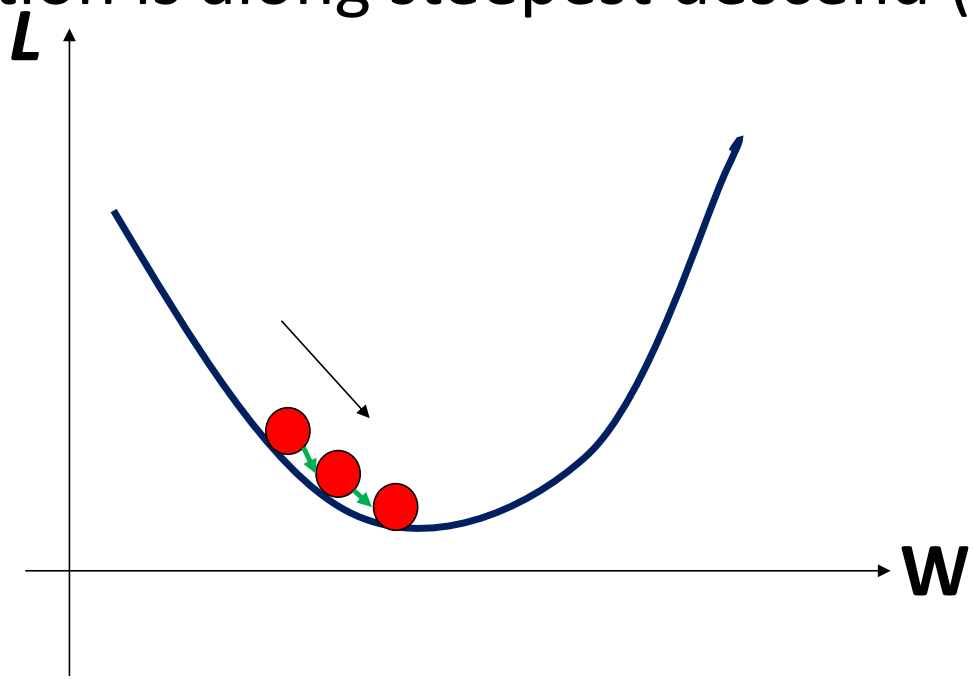
```
W = np.random.randn(10, 3073) * 0.001 # generate random starting W
bestloss = float("inf")
for i in xrange(1000):
  step_size = 0.0001
  Wtry = W + np.random.randn(10, 3073) * step_size
  loss = L(Xtr_cols, Ytr, Wtry)
  if loss < bestloss:
    W = Wtry
    bestloss = loss
  print 'iter %d loss is %f' % (i, bestloss)
```

Random search for update value

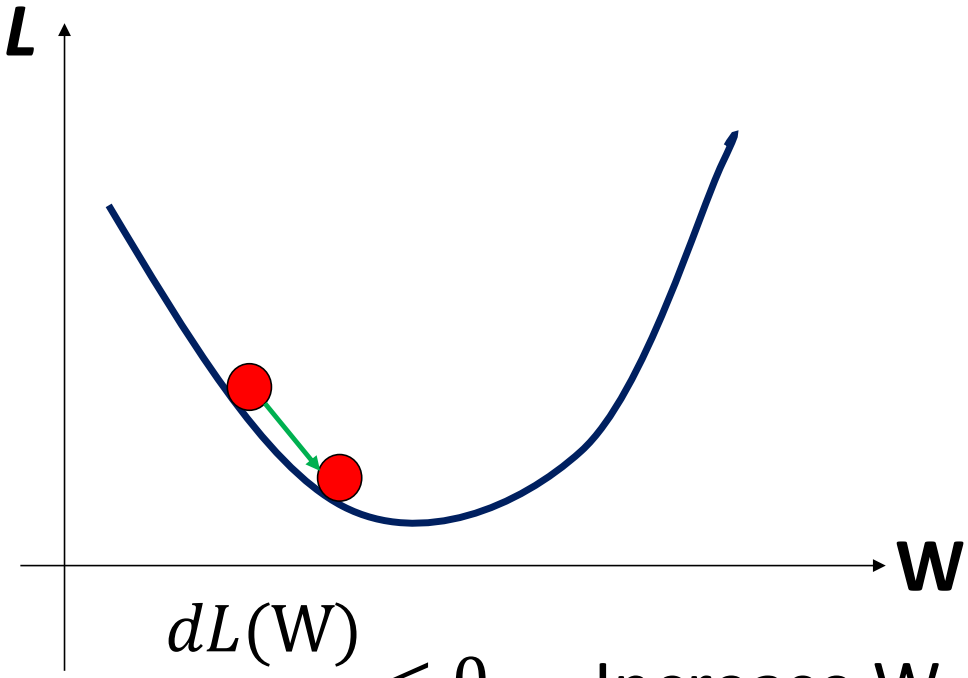
Update weight if loss decrease

Strategy #3: Following the Gradient (good)

- Mathematically, there is no need to randomly search
- Best direction is along steepest descend (via gradient)

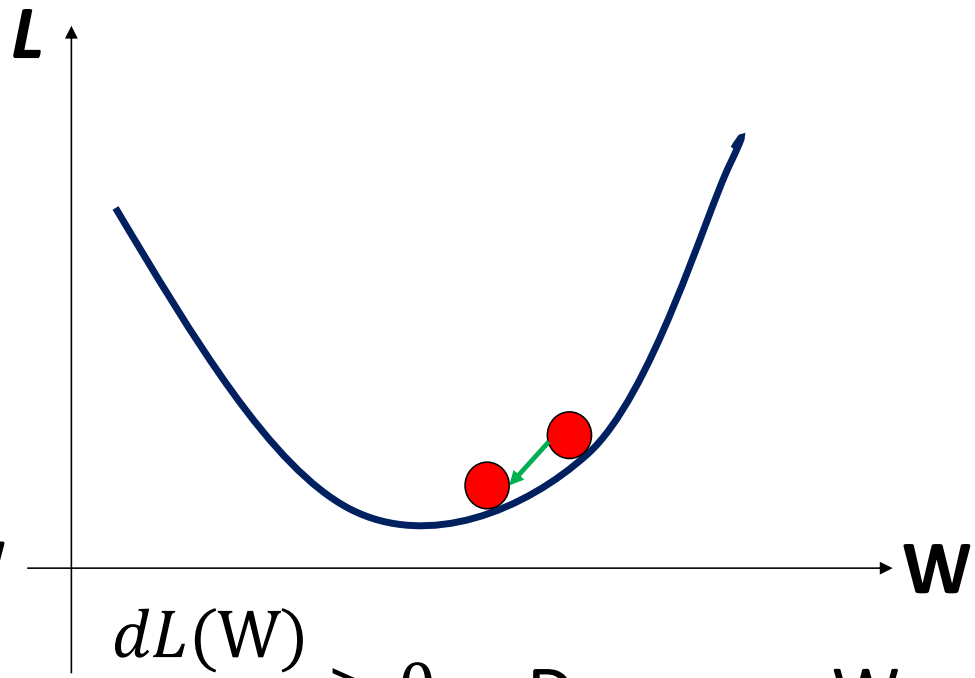


Update via Opposite Direction of Gradient



$$\frac{dL(\mathrm{W})}{d\mathrm{W}} < 0$$

Increase W



$$\frac{dL(\mathrm{W})}{d\mathrm{W}} > 0$$

Decrease W

Multiple Dimension

- In 1-dimension, gradient is the derivative of a function:

$$\frac{dL(w)}{dw} = \lim_{h \to 0} \frac{f(w+h) - f(w)}{h}$$

- In multiple-dimension, the gradient is the vector of partial derivatives:

$$\nabla_W L = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \dots, \frac{\partial L}{\partial w_n}]$$

Use Gradient to Update Weight

$$W \leftarrow W - \varepsilon \nabla_W L$$

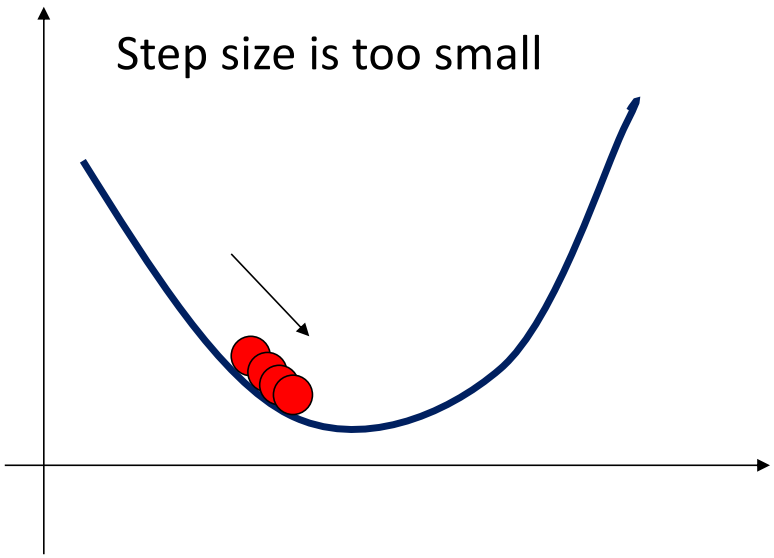
ε is a small step size (hyperparameter)

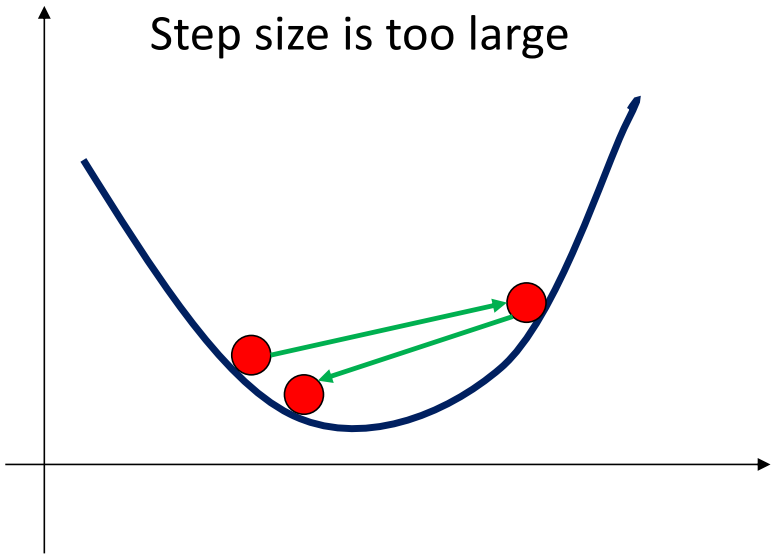
```
while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Update along opposite direction of gradient

Proper Step Size is Important

- Too small: Long training time
- Too big: Skip optimal point (hard to converge)





Compute Gradient (Numerical Method)

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

W + h (second dim):

[0.34,
-1.11 + **0.0001**,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25353

gradient dW:

[-2.5,
0.6,
?,
?,
?,…]

(1.25353 - 1.25347)/0.0001
= 0.6

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```python
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh = f(x) # evalute f(x + h)
    x[ix] = old_value # restore to previous value (very important!)

    # compute the partial derivative
    grad[ix] = (fxh - fx) / h # the slope
    it.iternext() # step to next dimension

  return grad
```

Small increment

New f(x) when update x in one dimension

Calculate partial derivative in that dimension

Two-sided Numerical Method

- For small values of *h,* two-sided estimation is more accurate approximation to the tangent line than the one-sided estimation.

$$\frac{dL(w)}{dw} = \lim_{h \to 0} \frac{f(w+h) - f(w-h)}{2h}$$

Mini-batch Stochastic Gradient Descent (SGD)

- Vanilla gradient descent need input all training data
  -- High memory and computation
- Mini-batch method split data to batches
  -- Update weight for each batch

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Notes on Mini-batch SGD

- Sometimes the name confuses
- Shuffling data before each epoch
- Good balance between SGD and batch GD

  -- (not too) fast learning, (not too) much memory
- It introduces noise, but noise helps

  -- avoid local minima
- Batch size is an (important) hyperparameter

Compute Gradient (Analytic Gradient)

- Loss function L is a function of W
- Gradient $\nabla_W L$ can be calculated via calculus
- Based on "Backpropagation" (next lecture)

$$L_{softmax} = \frac{1}{N}\sum_{i=1}^{N}\left(-log\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) + \lambda R(W)$$

$$w_i = w_i - step_{size} * \boxed{\frac{\partial L}{\partial w_i}}$$

Use calculus to compute

Analytic Method is Fast

current W:

[0.34,
-1.11,
0.78,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]

loss 1.25347

dW = ...
(some function
data and W)

gradient dW:

[-2.5,
0.6,
0,
0.2,
0.7,
-0.5,
1.1,
1.3,
-2.1,…]

Numerical vs Analytic Gradient

- Numerical gradient:

  -- Approximate, slow, easy to write
- Analytic gradient:

  -- Exact, fast, error-prone
- **Gradient check**: In practice, always use analytic gradient, but check implementation with numerical gradient

Acknowledgement

Many materials of the slides of this course are adopted and re-produced from several deep learning courses and tutorials.

- Prof. Fei-fei Li, Stanford, CS231n: Convolutional Neural Networks for Visual Recognition (online available)
- Prof. Andrew Ng, Stanford, CS230: Deep learning (online available)
- Prof. Yanzhi Wang, Northeastern, EECE7390: Advance in deep learning
- Prof. Jianting Zhang, CUNY, CSc G0815 High-Performance Machine Learning: Systems and Applications
- Prof. Vivienne Sze, MIT, “Tutorial on Hardware Architectures for Deep Neural Networks”
- Pytorch official tutorial https://pytorch.org/tutorials/