CS5670: Computer Vision

Noah Snavely

Optimization for machine learning

Slides from Fei-Fei Li, Justin Johnson, Serena Yeung
http://vision.stanford.edu/teaching/cs231n/

Readings

- Image classification:
  - http://cs231n.github.io/classification/
- Linear classification and loss functions:
  - http://cs231n.github.io/linear-classify/
- Optimization
  - http://cs231n.github.io/optimization-1/
  - http://cs231n.github.io/optimization-2/

Announcements

- Project 4 (Stereo) is out, due Thursday, April 26, 2018, by 11:59pm
  - To be done in groups of two
- Quiz 3 in class, Monday, 4/30, first 10 minutes of class
- Final exam in class, May 9

The story so far

$$s = f(x; W) = Wx$$

scores function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

SVM loss

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

data loss + regularization

We also learned about other data losses, e.g. the “softmax” loss

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

cat	**3.2**
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$

Softmax Function

cat	**3.2**
car	5.1
frog	-1.7

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$

Softmax Function

Probabilities must be >= 0

cat	**3.2**		**24.5**
car	5.1	$\xrightarrow{\exp}$	164.0
frog	-1.7		0.18

unnormalized probabilities

Softmax Classifier (Multinomial Logistic Regression)

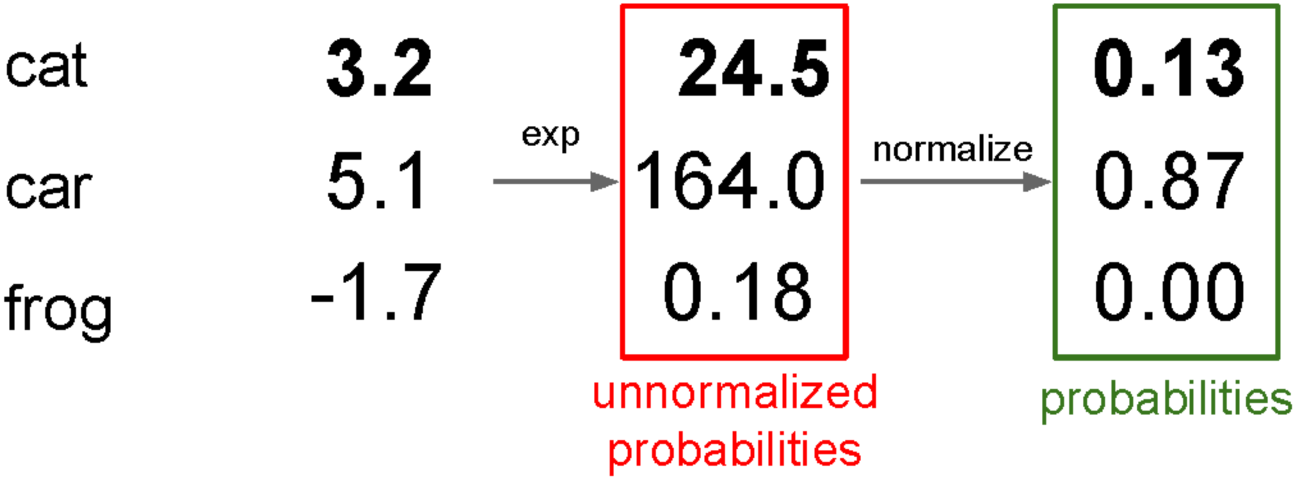
Want to interpret raw classifier scores as **probabilities**

$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$ Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1



		exp		normalize	
cat	**3.2**		**24.5**		**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7		0.18		0.00

unnormalized probabilities

probabilities

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

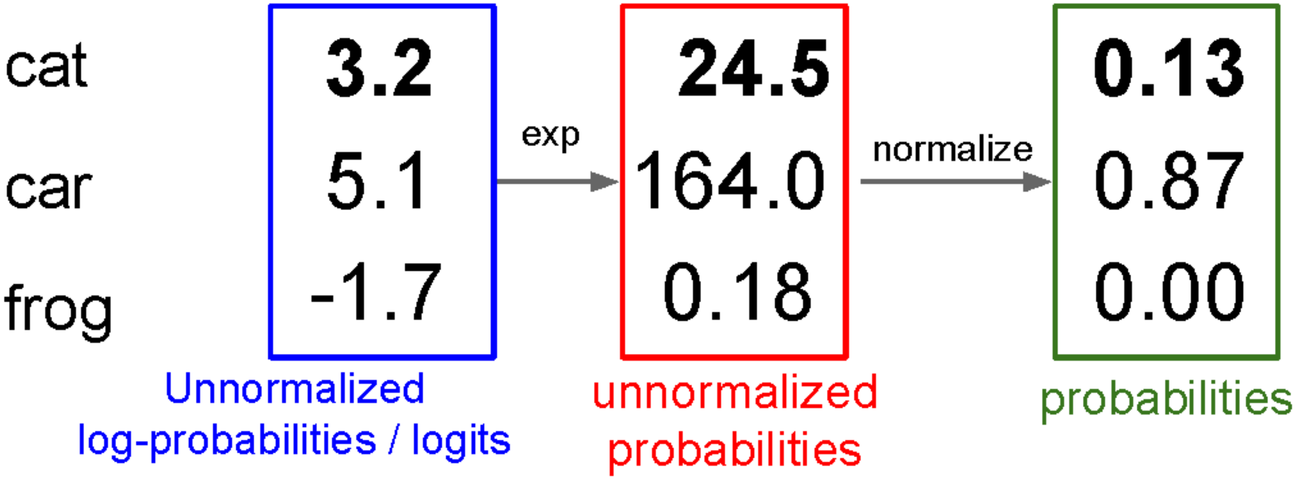
$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1



	Unnormalized log-probabilities / logits	unnormalized probabilities	probabilities
cat	**3.2**	**24.5**	**0.13**
car	5.1	164.0	0.87
frog	-1.7	0.18	0.00

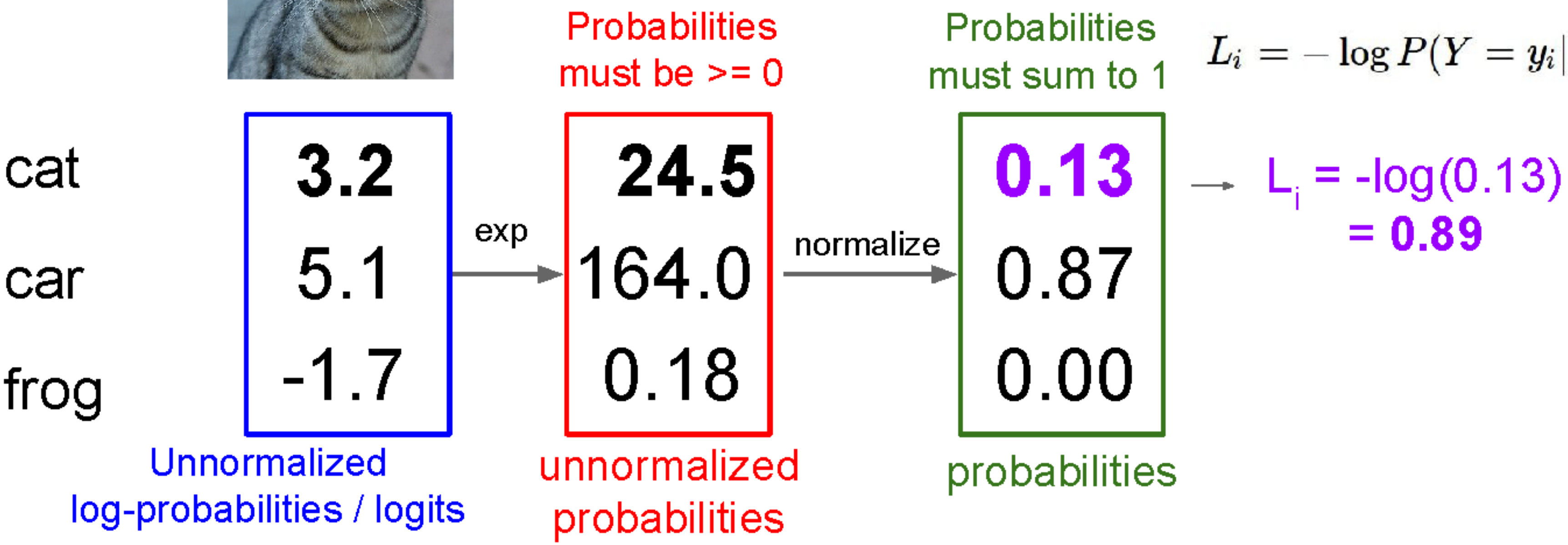
Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$ Softmax Function

$$L_i = -\log P(Y = y_i|X = x_i)$$



	Unnormalized log-probabilities / logits	unnormalized probabilities	probabilities
cat	**3.2**	**24.5**	**0.13**
car	5.1	164.0	0.87
frog	-1.7	0.18	0.00

→ $L_i = -\log(0.13)$
$= \mathbf{0.89}$

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

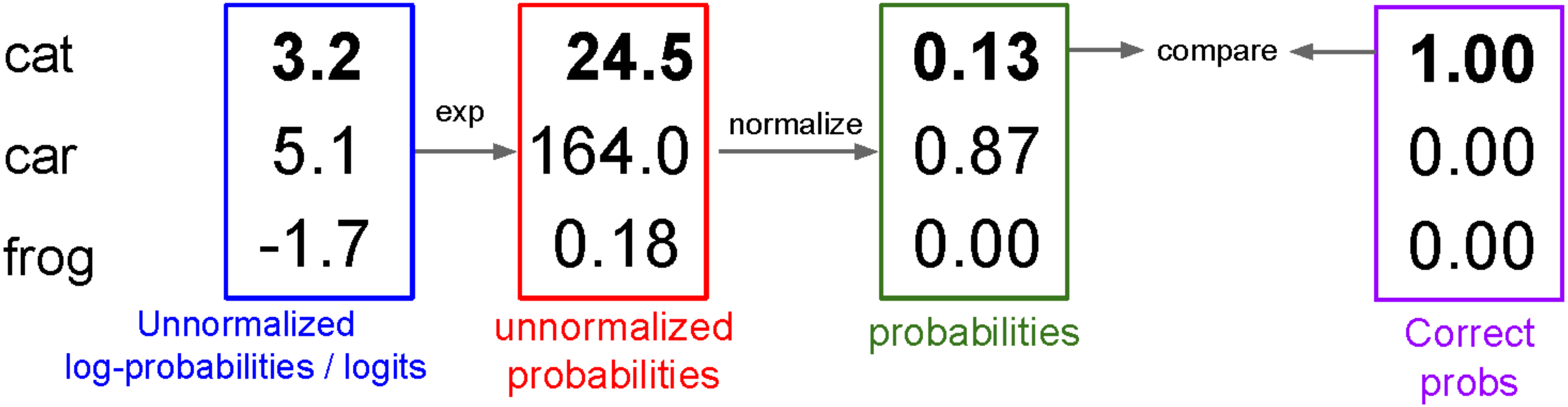
$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

$$L_i = -\log P(Y = y_i | X = x_i)$$



		exp		normalize		compare	
cat	**3.2**		**24.5**		**0.13**		**1.00**
car	5.1		164.0		0.87		0.00
frog	-1.7		0.18		0.00		0.00

Unnormalized log-probabilities / logits

unnormalized probabilities

probabilities

Correct probs

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

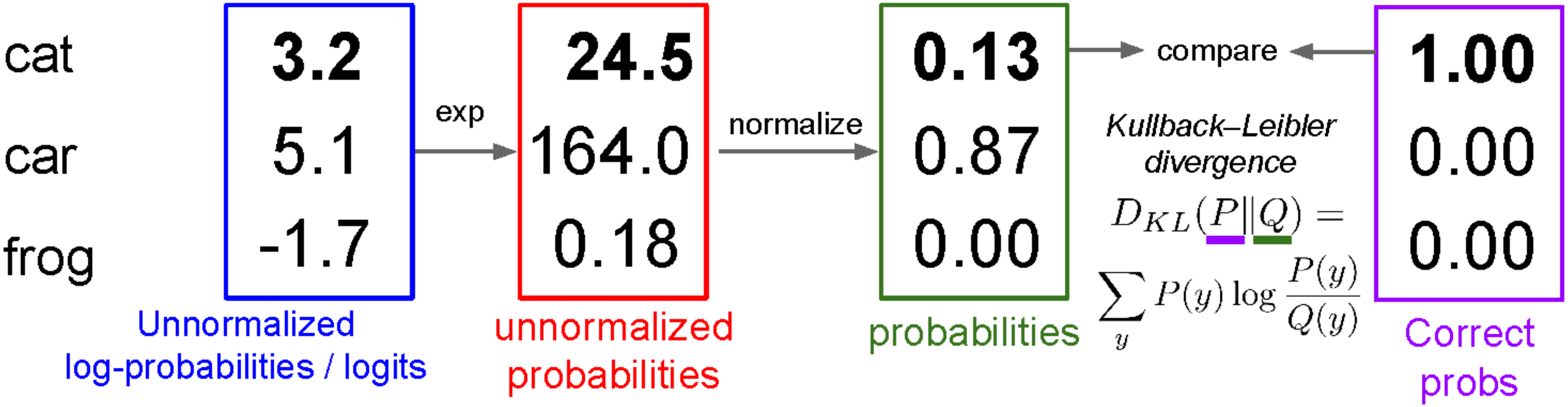
$$\boxed{s = f(x_i; W)}$$

$$\boxed{P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$ Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

$$L_i = -\log P(Y = y_i | X = x_i)$$



cat	**3.2**	**24.5**	**0.13**	**1.00**
car	5.1	164.0	0.87	0.00
frog	-1.7	0.18	0.00	0.00

Kullback–Leibler divergence

$$D_{KL}(\underline{P}\|\underline{Q}) = \sum_y P(y) \log \frac{P(y)}{Q(y)}$$

Unnormalized log-probabilities / logits

unnormalized probabilities

probabilities

Correct probs

Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as **probabilities**

$$\boxed{s = f(x_i; W)}$$

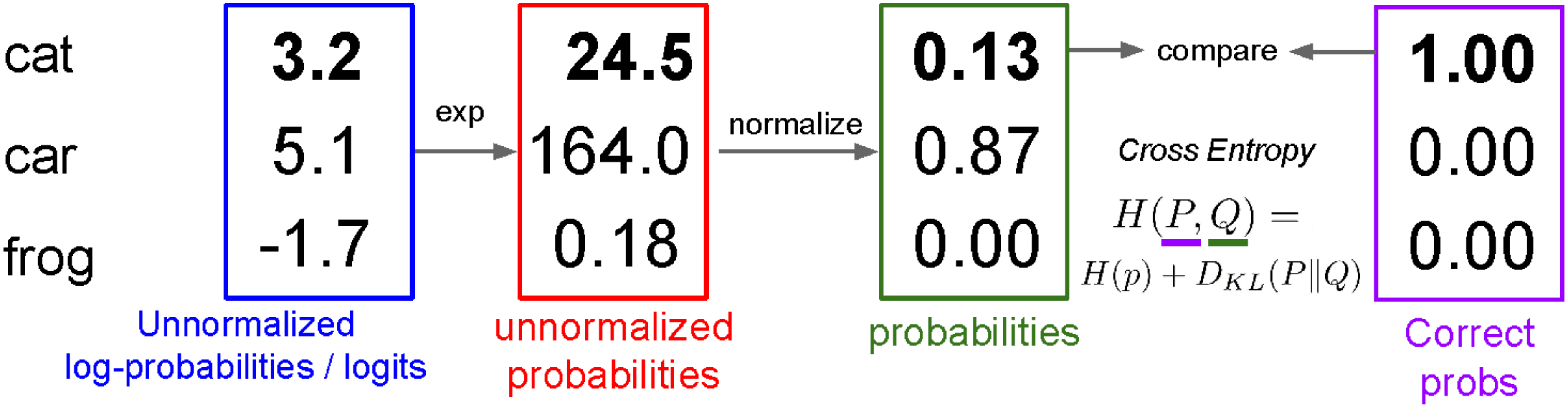
$$\boxed{P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}}$$

Softmax Function

Probabilities must be >= 0

Probabilities must sum to 1

$$L_i = -\log P(Y = y_i|X = x_i)$$



	Unnormalized log-probabilities / logits	unnormalized probabilities	probabilities	Correct probs
cat	**3.2**	**24.5**	**0.13**	**1.00**
car	5.1	164.0	0.87	0.00
frog	-1.7	0.18	0.00	0.00

exp

normalize

compare

Cross Entropy

$$H(\underline{P}, \underline{Q}) = H(p) + D_{KL}(P\|Q)$$

The story so far

$$s = f(x; W) = Wx$$

scores function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

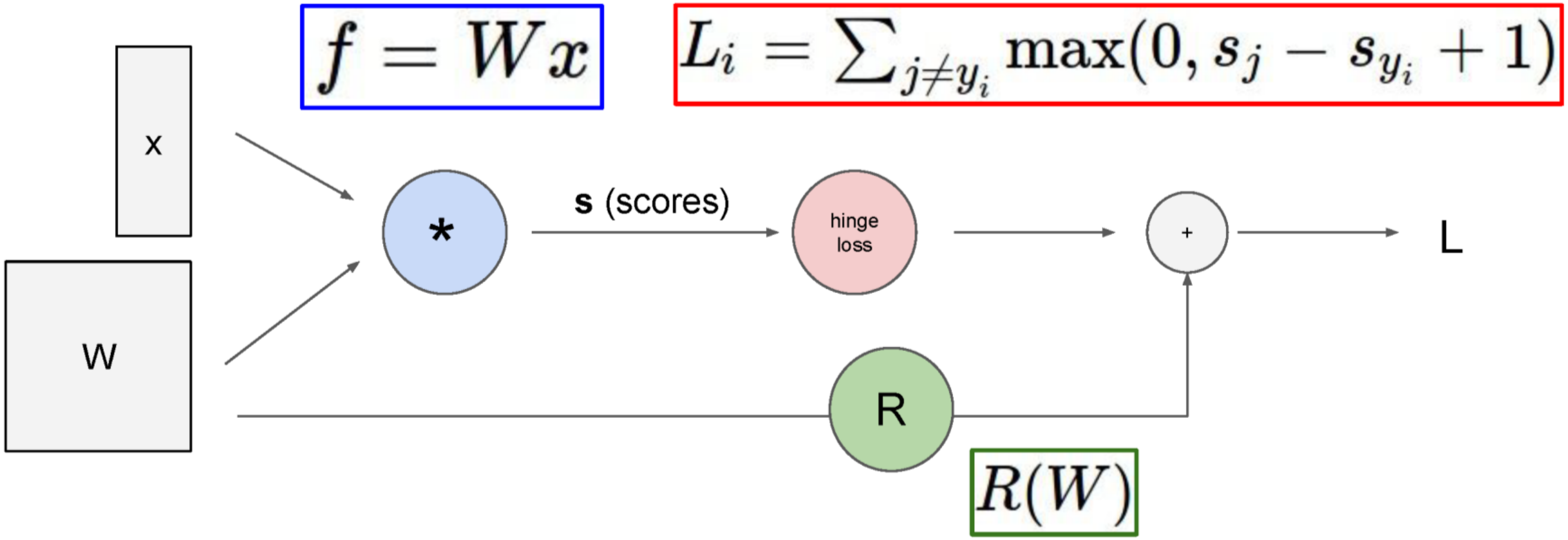
SVM loss

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

data loss + regularization

We also learned about other data losses, e.g. the “softmax” loss

Computation graphs



$$\boxed{f = Wx}$$

$$\boxed{L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)}$$

$$\boxed{R(W)}$$

Convolutional network (AlexNet)

input image

weights

loss

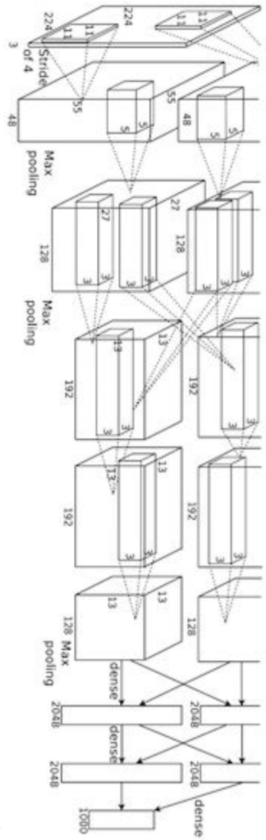


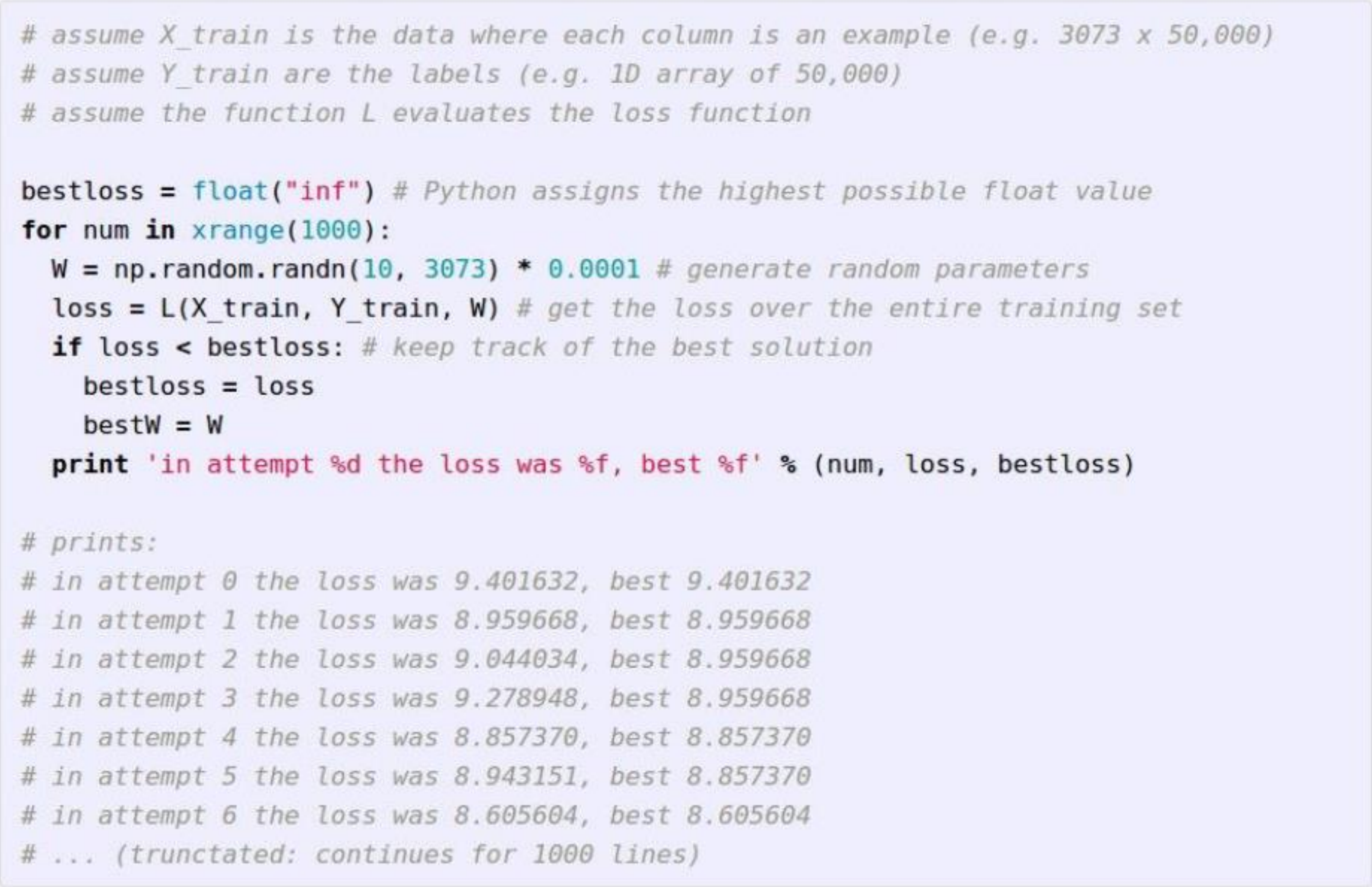
Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

(a much bigger computation graph)

How do we set the weights?

- Need to solve an optimization problem:
  - Find the weights *W* that minimize the training loss *L*
- In general this is a non-linear, non-convex problem
  - Closed-form solvers do not generally exist, unlike with e.g. least squares problems
  - Might not find the globally optimal weights
- (Side note: some learning problems, such as linear SVMs, do have convex loss functions)

Strategy #1: A bad idea: **Random search**

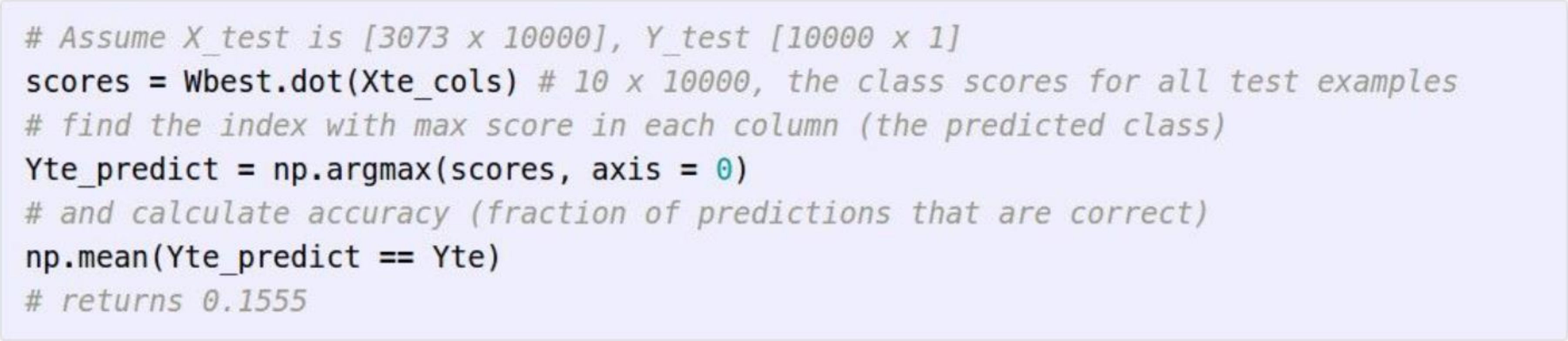


```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

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)
```

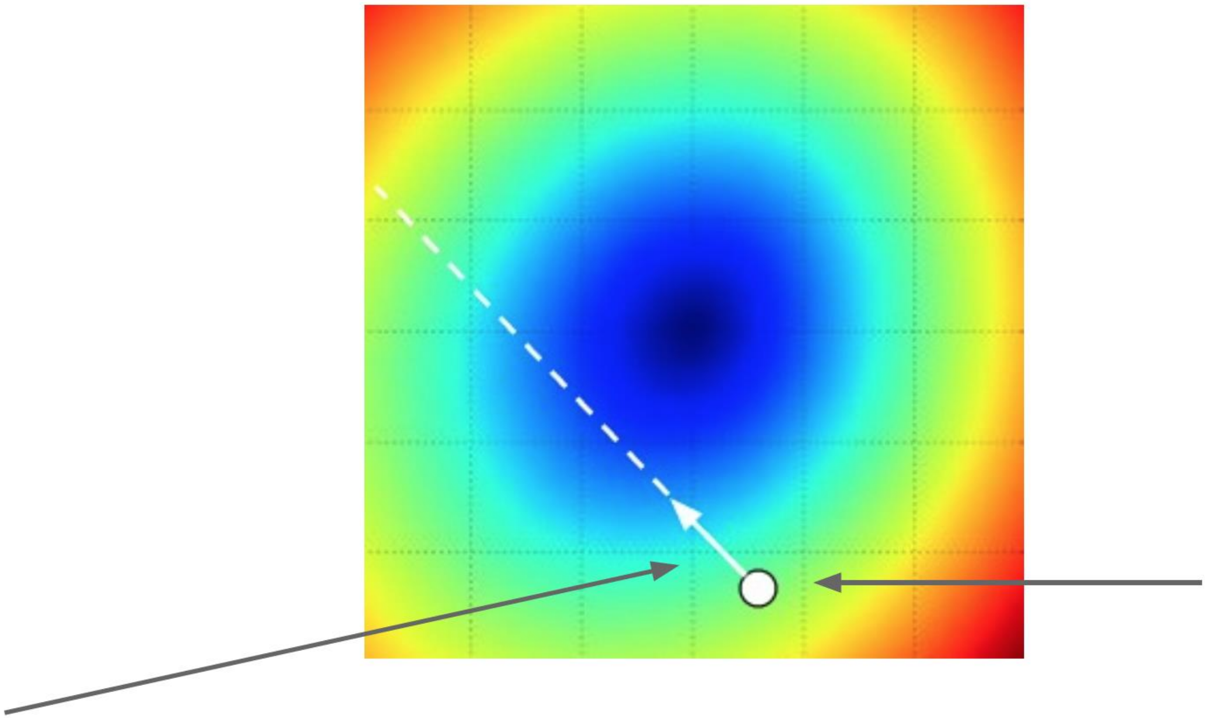
Lets see how well this works on the test set...



```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!
(SOTA is ~95%)

Strategy #2: Follow the slope (aka Gradient Descent)



original W

negative gradient direction

Gradient descent: walk in the direction opposite gradient

- **Q**: How far?
- **A**: Step size: *learning rate*
- Too big: will miss the minimum
- Too small: slow convergence

Strategy #2: **Follow the slope**

In 1-dimension, the derivative of a function:

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

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient
The direction of steepest descent is the **negative gradient**

current W:

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

loss 1.25347

gradient dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

current W:	**W + h** (first dim)**:**	**gradient dW:**
[0.34,	[0.34 + **0.0001**,	[?,
-1.11,	-1.11,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25322**	

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 (first dim)**:**

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

loss 1.25322

gradient dW:

[**-2.5**,
?,
?,

(1.25322 - 1.25347)/0.0001
= -2.5

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

?,
?,…]

current W:	**W + h** (second dim)**:**	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	?,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25353**	

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

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

?,…]

current W:	**W + h** (third dim):	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11,	0.6,
0.78,	0.78 + **0.0001**,	?,
0.12,	0.12,	?,
0.55,	0.55,	?,
2.81,	2.81,	?,
-3.1,	-3.1,	?,
-1.5,	-1.5,	?,
0.33,…]	0.33,…]	?,…]
loss 1.25347	**loss 1.25347**	

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 (third dim):

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?
?,…]

(1.25347 - 1.25347)/0.0001
= 0

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

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 (third dim)**:**

[0.34,
-1.11,
0.78 + **0.0001**,
0.12,
0.55,
2.81,
-3.1,
-1.5,
0.33,…]
loss 1.25347

gradient dW:

[-2.5,
0.6,
0,
?,
?,
?,…]

Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

But the loss is just a function of W!

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

But the loss is just a function of W!

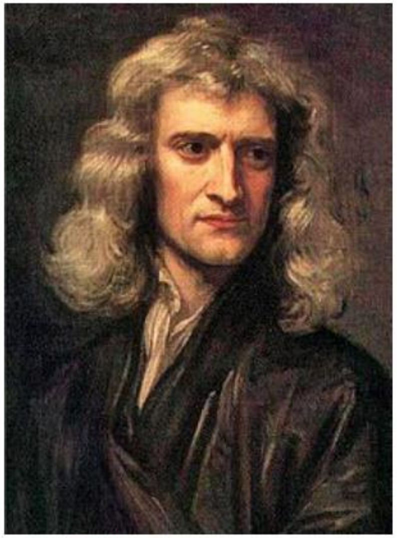
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Use calculus to compute an **analytic gradient**



This image is in the public domain

This image is in the public domain

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,…]

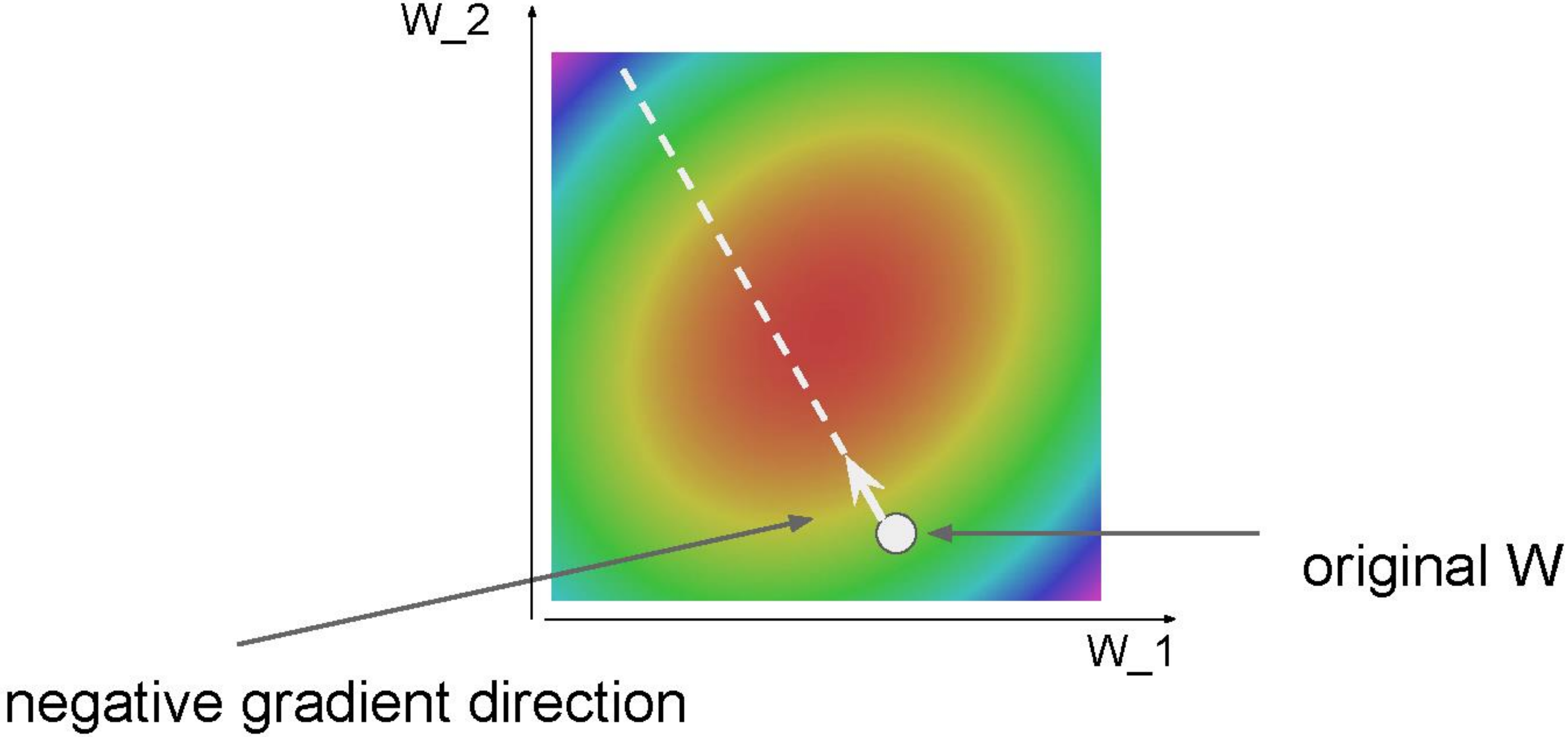
In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

=>

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

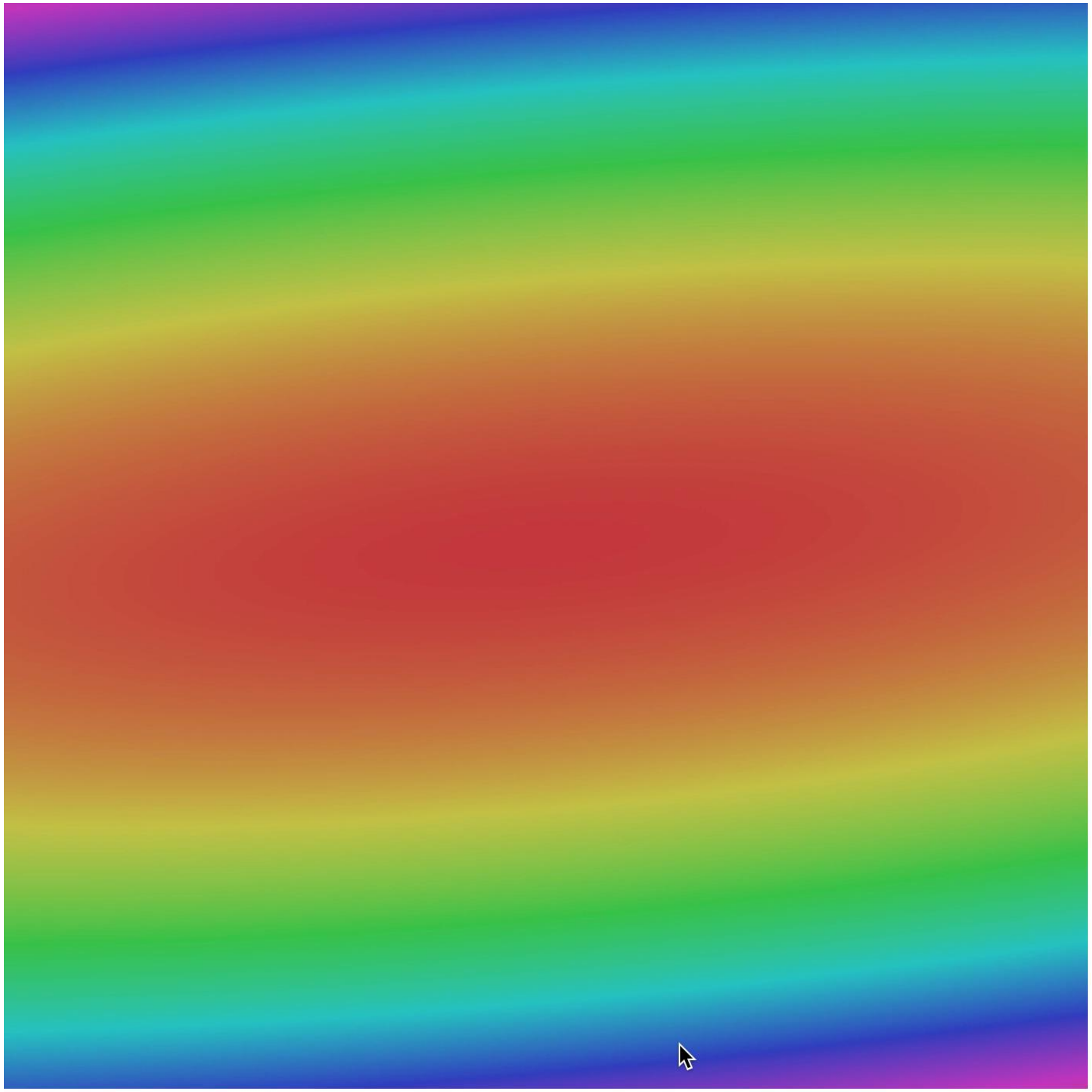
Questions?



original W

negative gradient direction

Gradient descent in action



Analytic Gradient

Single term of SVM (hinge) data loss:

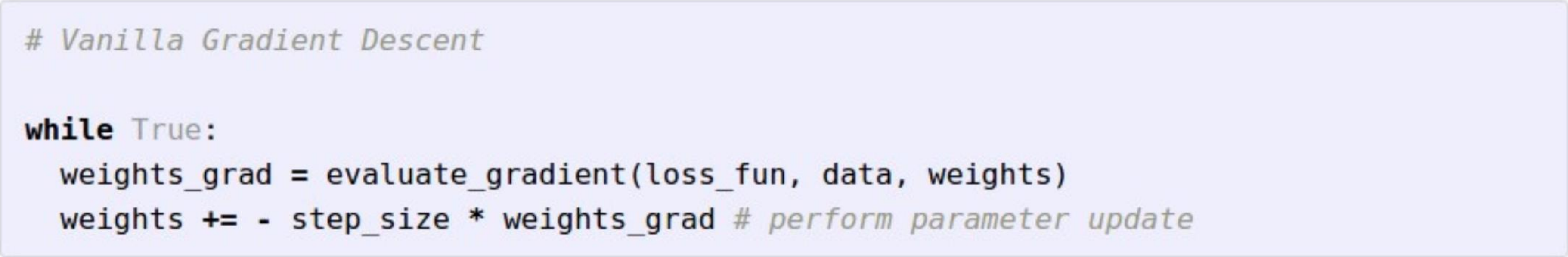
$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

$$\nabla_{w_{y_i}} L_i = - \left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i$$

Full gradient is the sum of all L_is over all training examples x_i

Gradient Descent



```
# Vanilla Gradient Descent

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

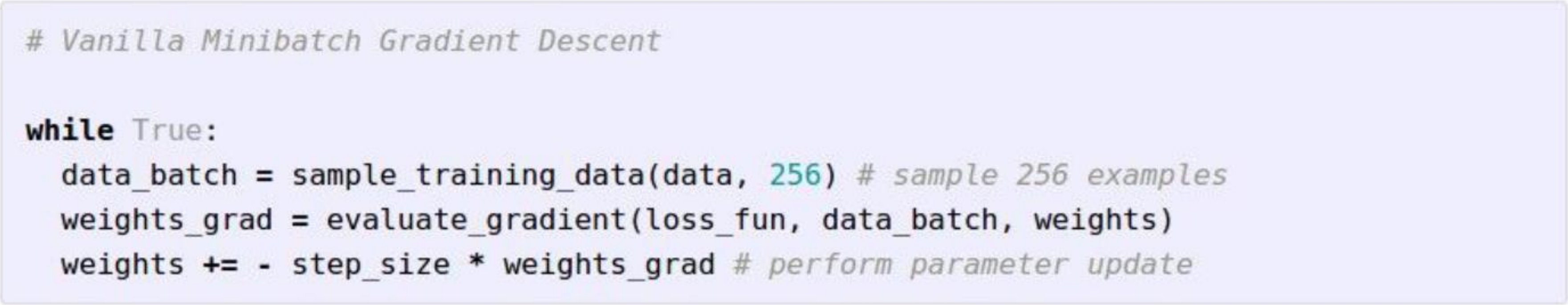
Stochastic Gradient Descent (SGD)

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

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

Full sum expensive when N is large!

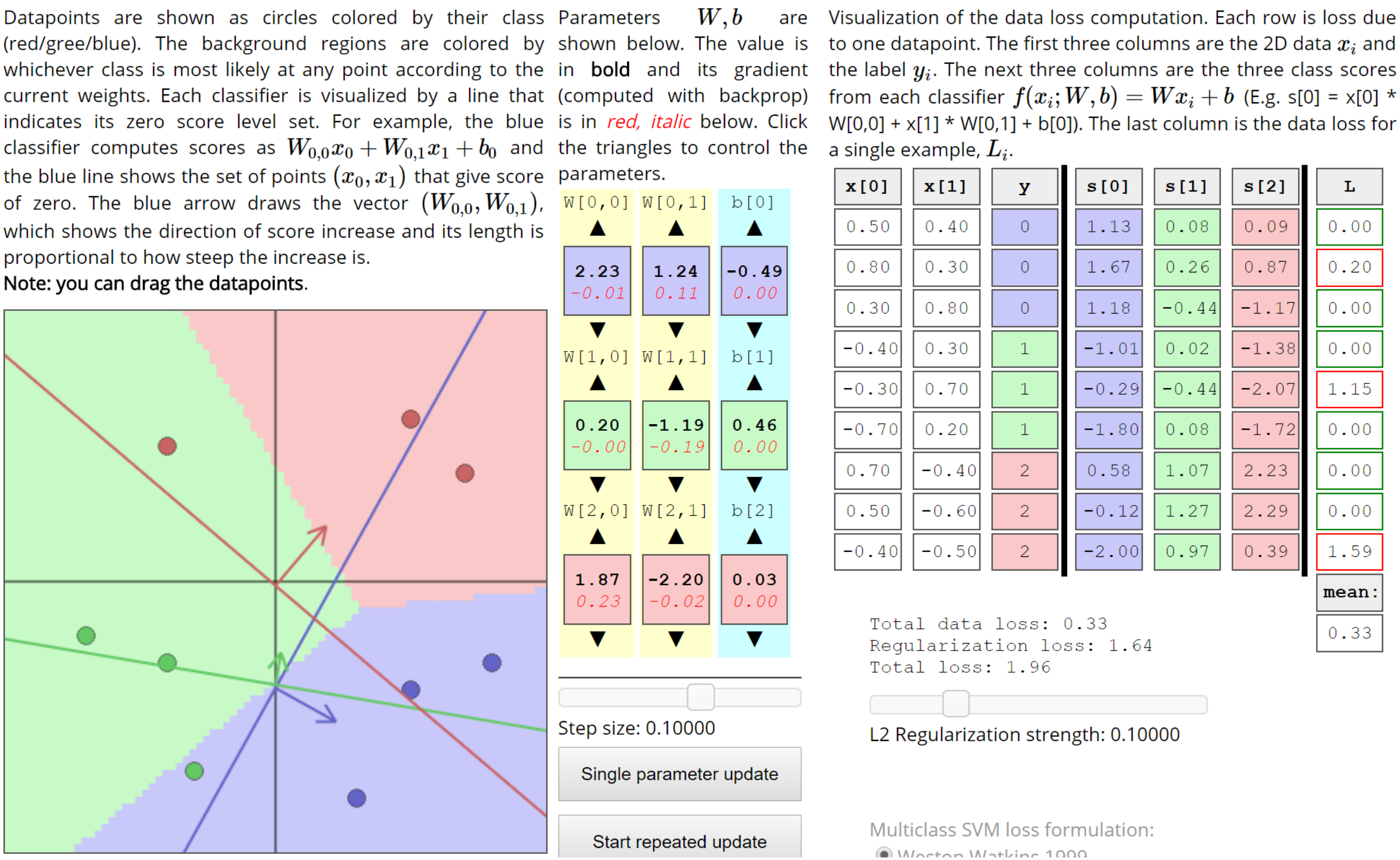
Approximate sum using a **minibatch** of examples
32 / 64 / 128 common



```
# 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
```

Interactive Web Demo



Datapoints are shown as circles colored by their class (red/gree/blue). The background regions are colored by whichever class is most likely at any point according to the current weights. Each classifier is visualized by a line that indicates its zero score level set. For example, the blue classifier computes scores as $W_{0,0}x_0 + W_{0,1}x_1 + b_0$ and the blue line shows the set of points (x_0, x_1) that give score of zero. The blue arrow draws the vector $(W_{0,0}, W_{0,1})$, which shows the direction of score increase and its length is proportional to how steep the increase is.

Note: you can drag the datapoints.

Parameters W, b are shown below. The value is in **bold** and its gradient (computed with backprop) is in *red, italic* below. Click the triangles to control the parameters.

W[0,0]	W[0,1]	b[0]
▲	▲	▲
2.23 *-0.01*	**1.24** *0.11*	**-0.49** *0.00*
▼	▼	▼
W[1,0]	W[1,1]	b[1]
▲	▲	▲
0.20 *-0.00*	**-1.19** *-0.19*	**0.46** *0.00*
▼	▼	▼
W[2,0]	W[2,1]	b[2]
▲	▲	▲
1.87 *0.23*	**-2.20** *-0.02*	**0.03** *0.00*
▼	▼	▼

Step size: 0.10000

Single parameter update

Start repeated update

Visualization of the data loss computation. Each row is loss due to one datapoint. The first three columns are the 2D data x_i and the label y_i. The next three columns are the three class scores from each classifier $f(x_i; W, b) = Wx_i + b$ (E.g. s[0] = x[0] * W[0,0] + x[1] * W[0,1] + b[0]). The last column is the data loss for a single example, L_i.

x[0]	x[1]	y	s[0]	s[1]	s[2]	L
0.50	0.40	0	1.13	0.08	0.09	0.00
0.80	0.30	0	1.67	0.26	0.87	0.20
0.30	0.80	0	1.18	-0.44	-1.17	0.00
-0.40	0.30	1	-1.01	0.02	-1.38	0.00
-0.30	0.70	1	-0.29	-0.44	-2.07	1.15
-0.70	0.20	1	-1.80	0.08	-1.72	0.00
0.70	-0.40	2	0.58	1.07	2.23	0.00
0.50	-0.60	2	-0.12	1.27	2.29	0.00
-0.40	-0.50	2	-2.00	0.97	0.39	1.59
						mean:
						0.33

```
Total data loss: 0.33
Regularization loss: 1.64
Total loss: 1.96
```

L2 Regularization strength: 0.10000

Multiclass SVM loss formulation:

● Weston Watkins 1999

http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/

The dynamics of Gradient Descent

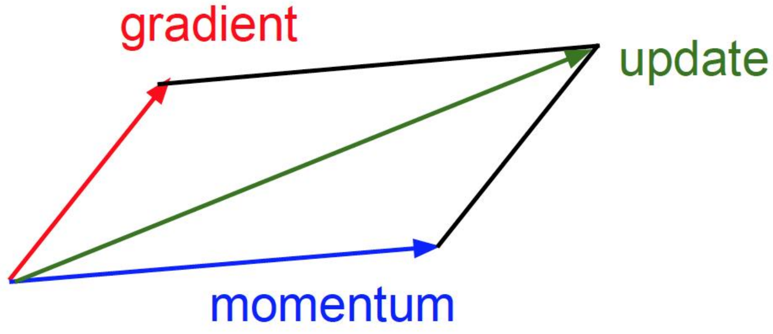
pull some weights up and some down

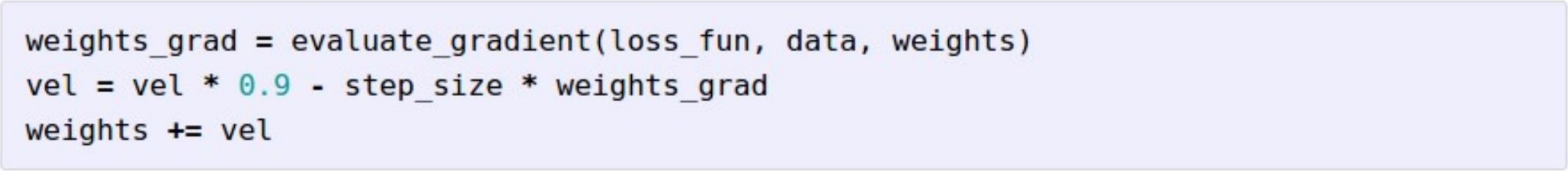
$$L = \boxed{\frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + \Delta)\right]} + \boxed{\lambda \sum_k \sum_l W_{k,l}^2}$$

$$L = \boxed{\frac{1}{N}\sum_i - \log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)} + \boxed{\lambda \sum_k \sum_l W_{k,l}^2}$$

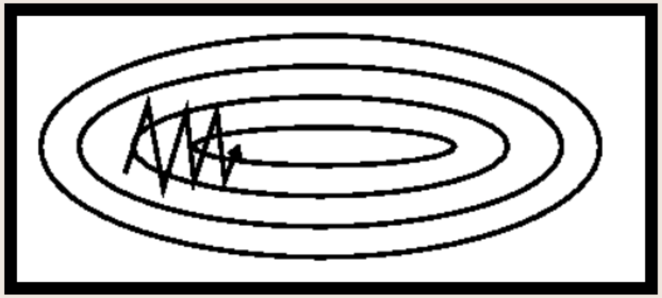
always pull the weights down

Momentum Update

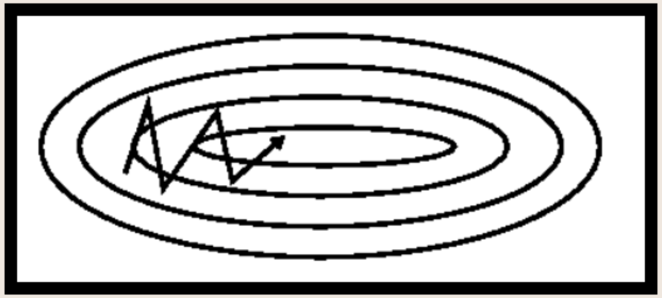




```
weights_grad = evaluate_gradient(loss_fun, data, weights)
vel = vel * 0.9 - step_size * weights_grad
weights += vel
```



(Fig. 2a)



(Fig. 2b)

Many other ways to perform optimization…

- Second order methods that use the Hessian (or its approximation): BFGS, **LBFGS**, etc.

- Currently, the lesson from the trenches is that well-tuned SGD+Momentum is very hard to beat for CNNs.

Questions?

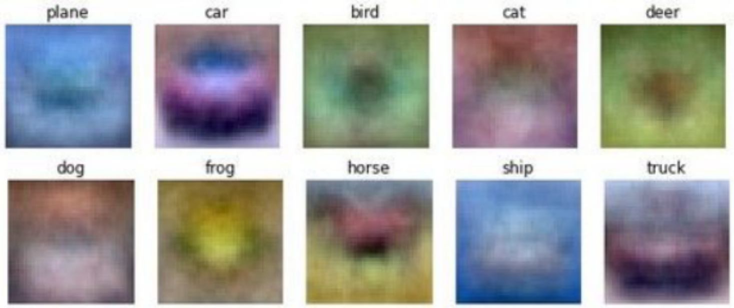
Where are we?

- Classifiers: SVM vs. Softmax
- Gradient descent to optimize loss functions
  - Batch gradient descent, stochastic gradient descent
  - Momentum
  - Numerical gradients (slow, approximate), analytic gradients (fast, error-prone)

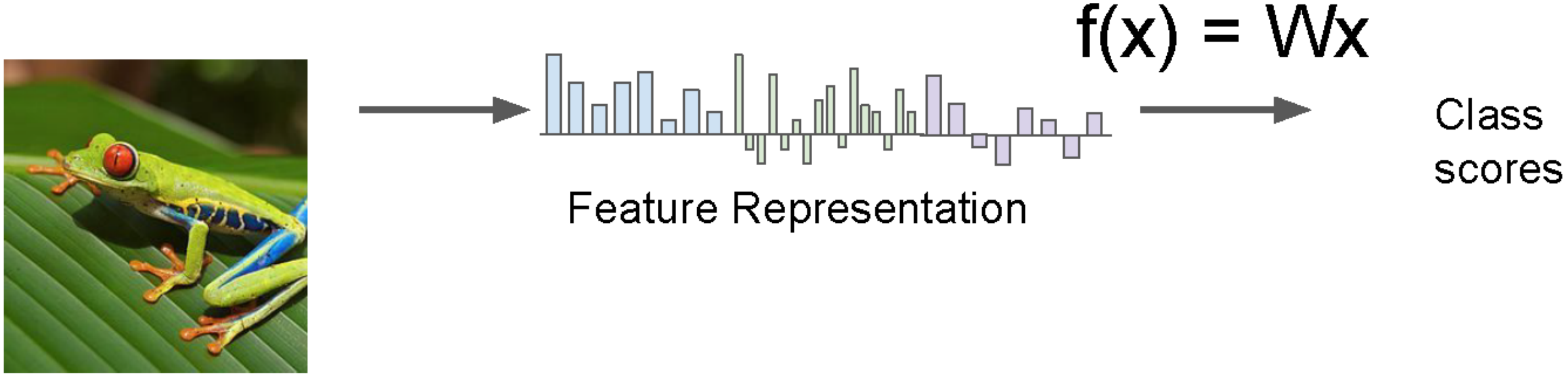
Aside: Image Features

Class scores

$$f(x) = Wx$$



Aside: Image Features

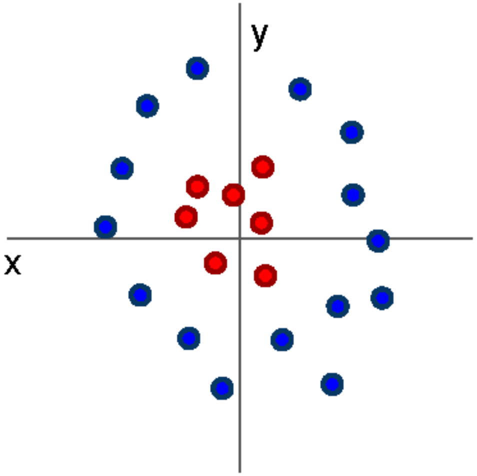


Feature Representation

$f(x) = Wx$

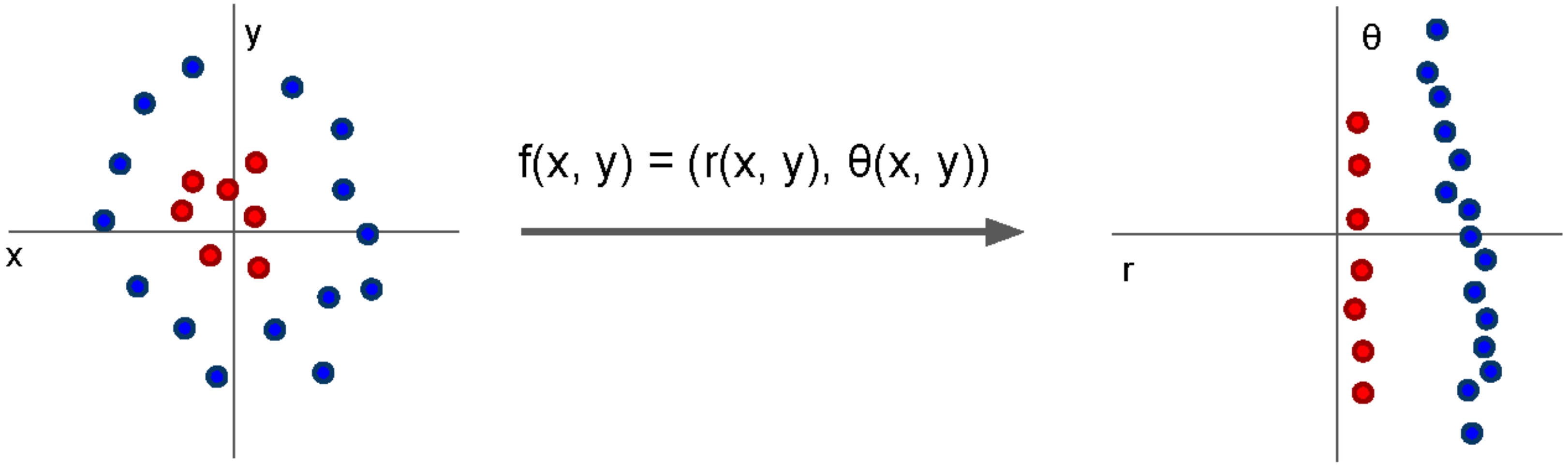
Class scores

Image Features: Motivation



Cannot separate red and blue points with linear classifier

Image Features: Motivation

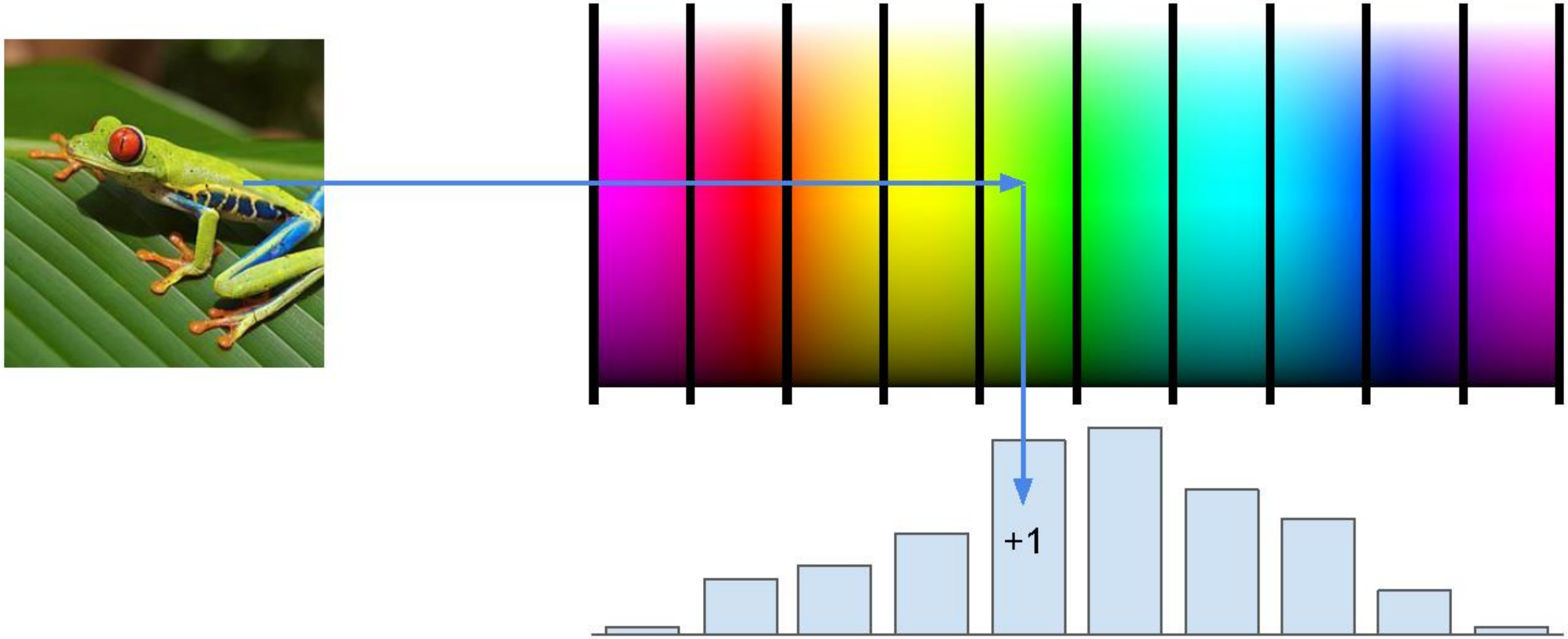


$f(x, y) = (r(x, y), \theta(x, y))$

Cannot separate red and blue points with linear classifier

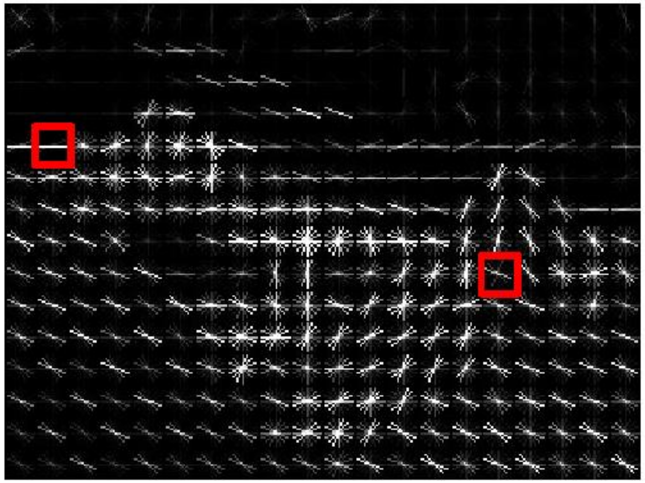
After applying feature transform, points can be separated by linear classifier

Example: Color Histogram



Example: Histogram of Oriented Gradients (HoG)

Divide image into 8x8 pixel regions
Within each region quantize edge direction into 9 bins

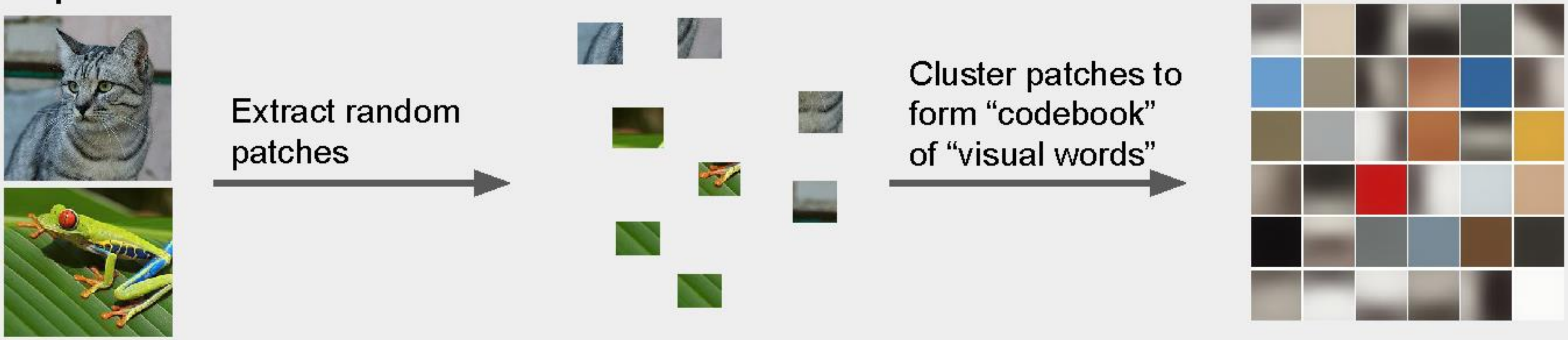


Example: 320x240 image gets divided into 40x30 bins; in each bin there are 9 numbers so feature vector has 30*40*9 = 10,800 numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

Example: Bag of Words

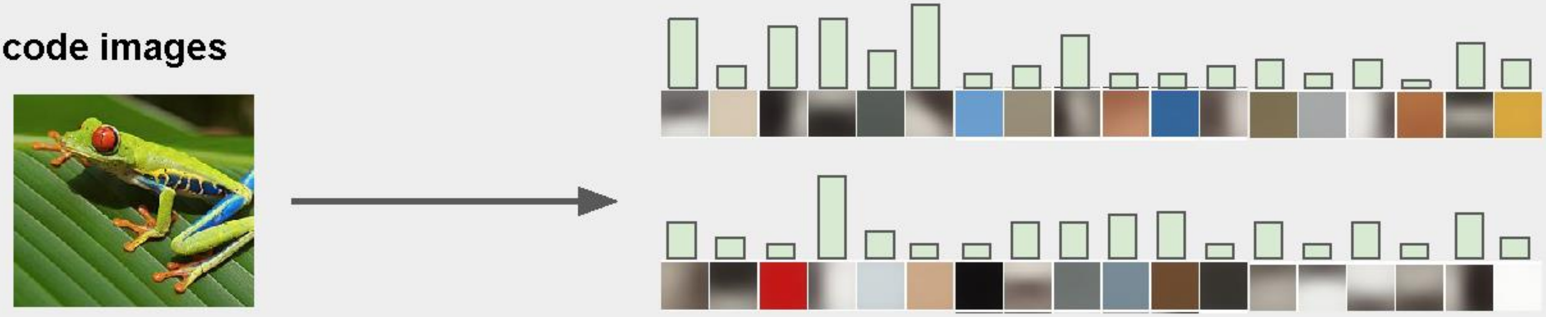
Step 1: Build codebook



Extract random patches

Cluster patches to form “codebook” of “visual words”

Step 2: Encode images



Fei-Fei and Perona, "A bayesian hierarchical model for learning natural scene categories", CVPR 2005

Aside: Image Features

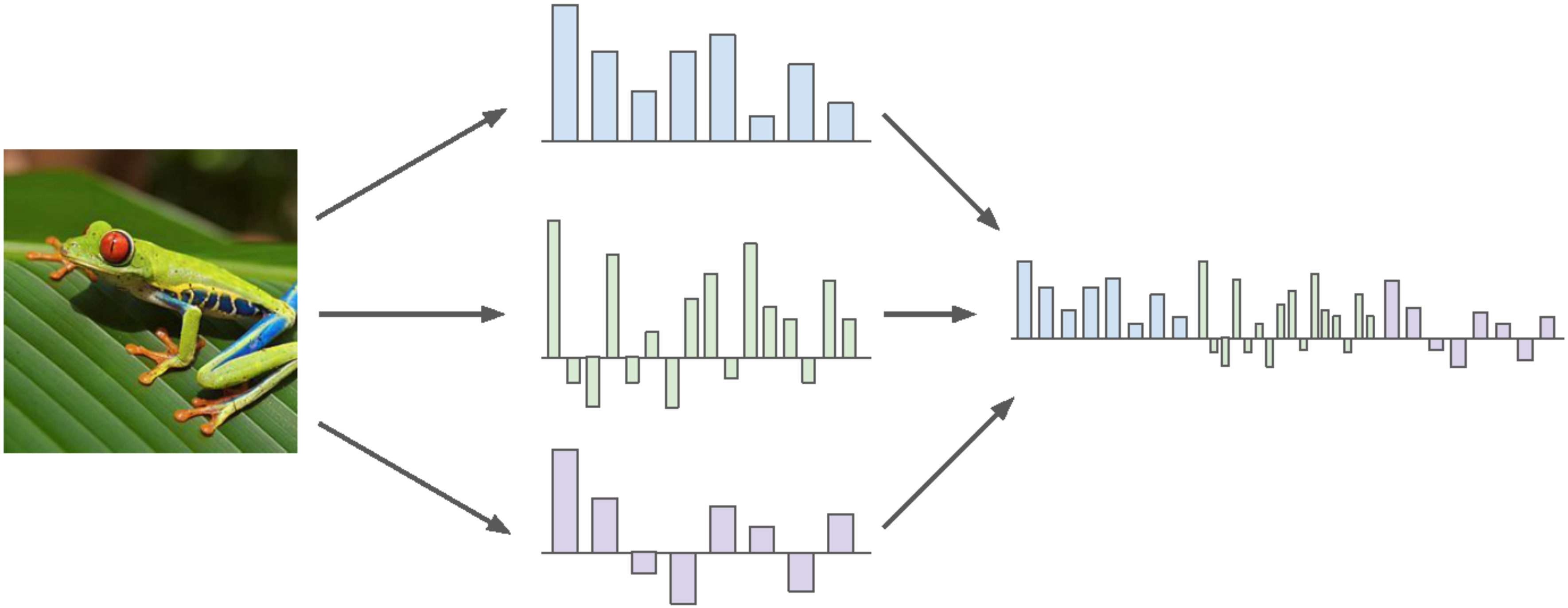
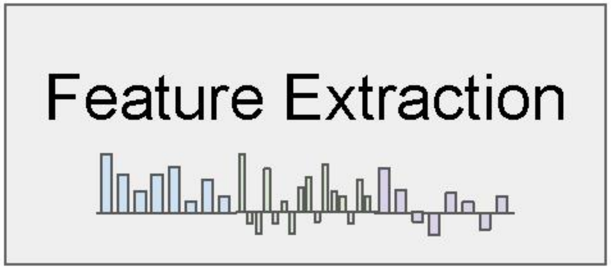


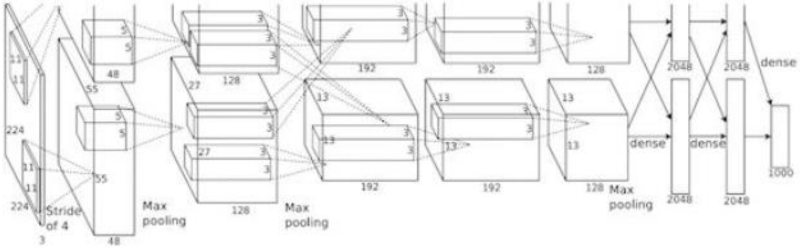
Image features vs ConvNets



f

10 numbers giving scores for classes

training



Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012. Figure copyright Krizhevsky, Sutskever, and Hinton, 2012. Reproduced with permission.

10 numbers giving scores for classes

training

Questions?

Next: Neural networks