Deep Learning

First classifier: **Nearest Neighbor**

```
def train(images, labels):
  # Machine learning!
  return model
```

Memorize all data and labels

```
def predict(model, test_images):
  # Use model to predict labels
  return test_labels
```

Predict the label of the most similar training image

Parametric Approach: Linear Classifier

$$f(x,W) = Wx$$

Image

Array of **32x32x3** numbers
(3072 numbers total)

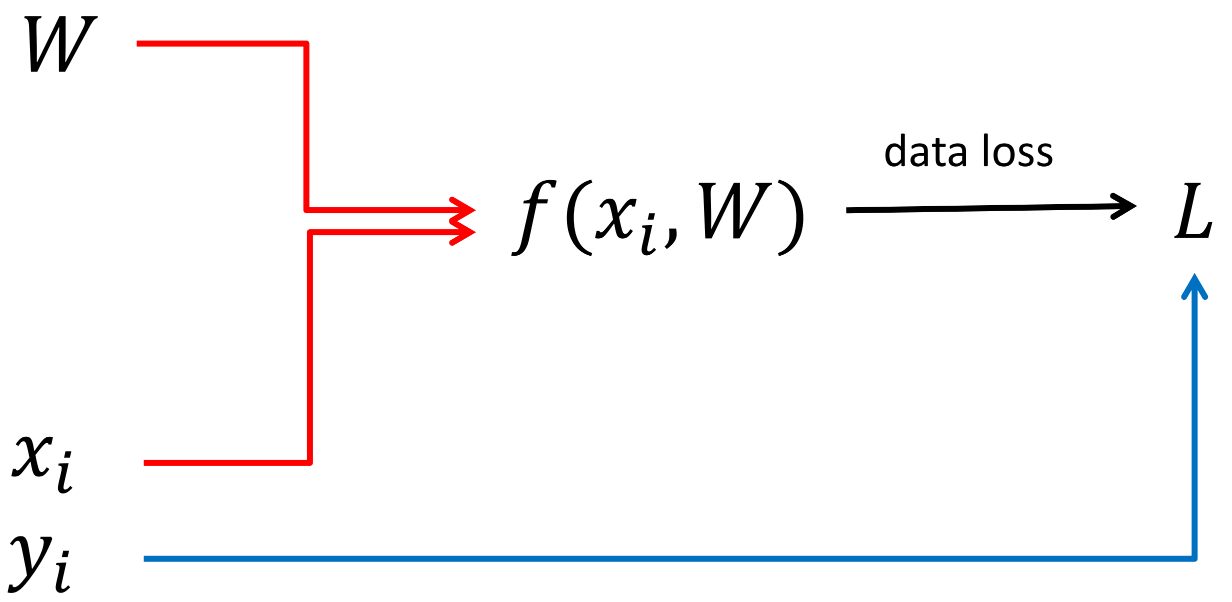
$f(\mathbf{x},\mathbf{W})$

10 numbers defining
class scores

W

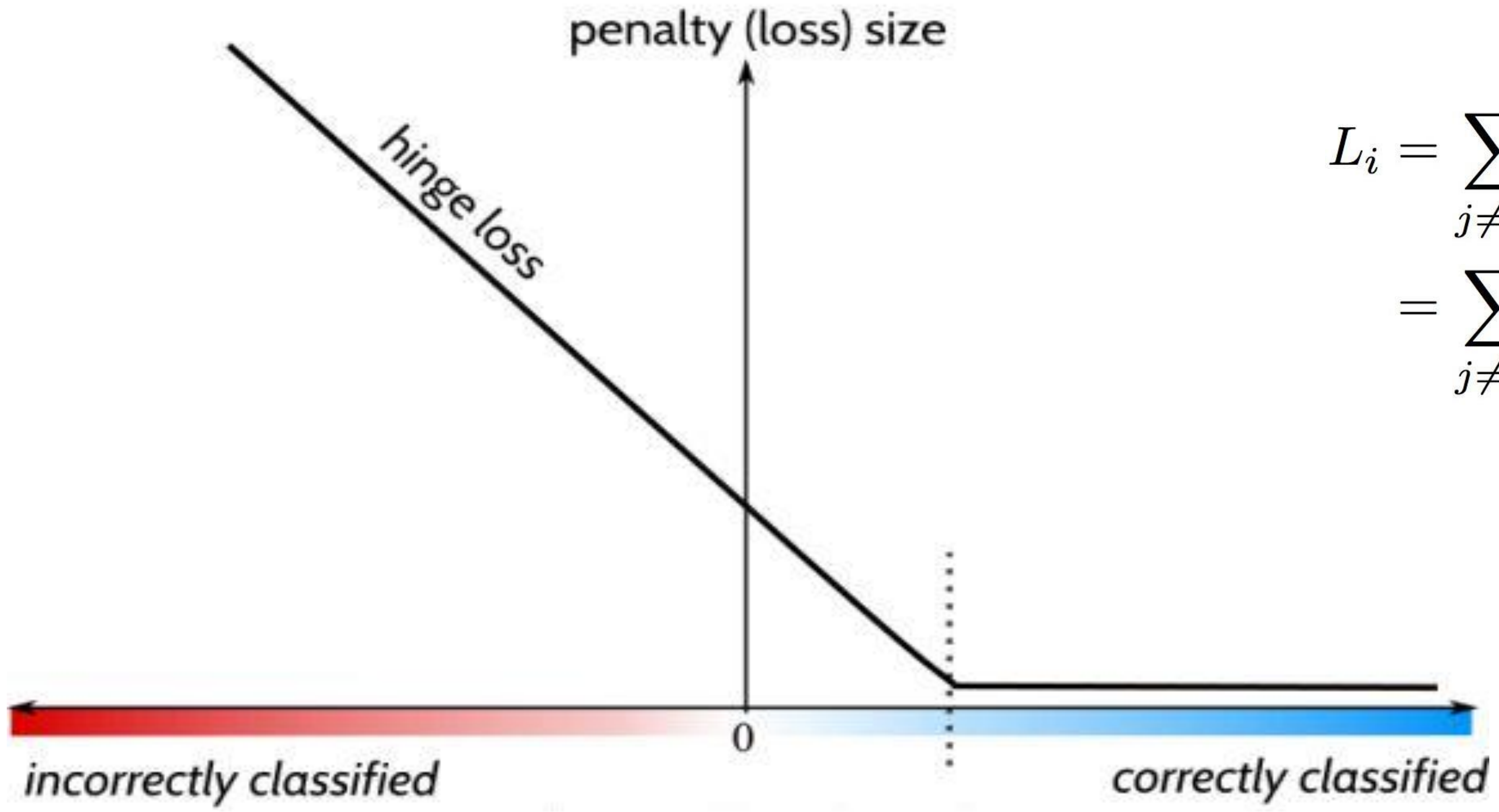
parameters
or weights

Loss Function



L: Metric to assess what loss of data classification our model incurs

Hinge loss



$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$
$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Softmax Classifier (Multinomial Logistic Regression)

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

where

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

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

$$P(Y=k|X=x_i) = \boxed{\frac{e^{s_k}}{\sum_j e^{s_j}}} \quad \text{where} \quad s = f(x_i; W)$$

Softmax function

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

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

where

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

scores

3.2 $\rightarrow$ $\rightarrow$ 0.2

5.1 $\rightarrow$ Softmax function $\rightarrow$ 0.8

-1.7 $\rightarrow$ $\rightarrow$ 0.0

probabilities
or confidence

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

$$P(Y=k|X=x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

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

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

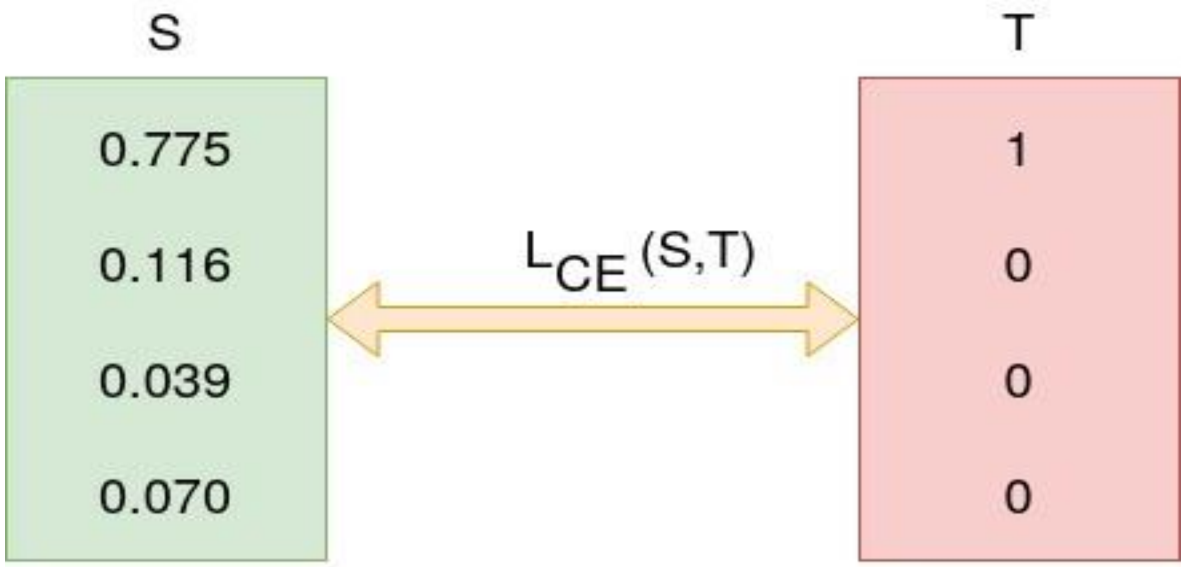
$$L_i = -\log P(Y = y_i|X = x_i)$$

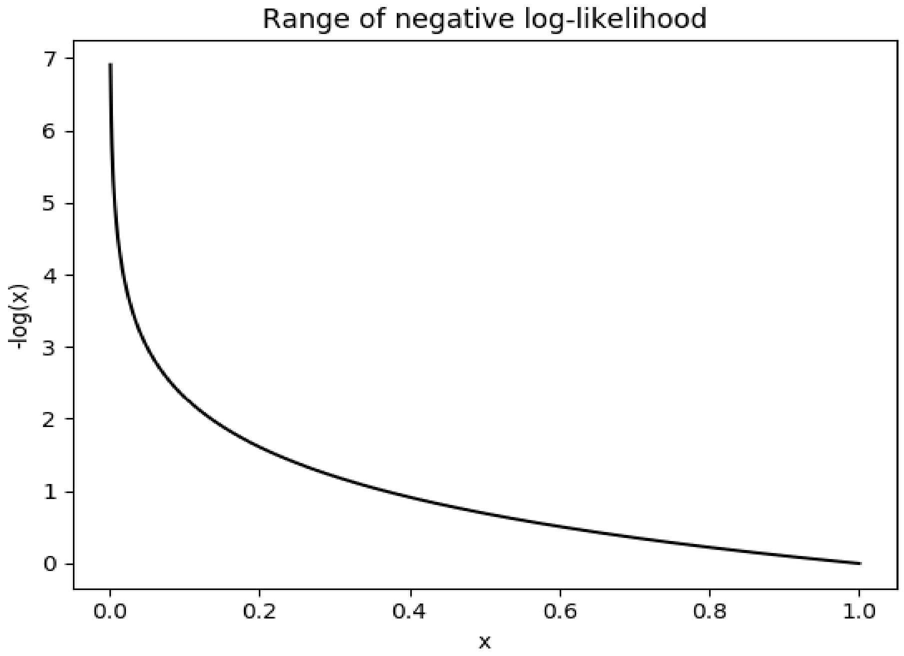
---

in summary: $$L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$$

Cross-Entropy Loss

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$





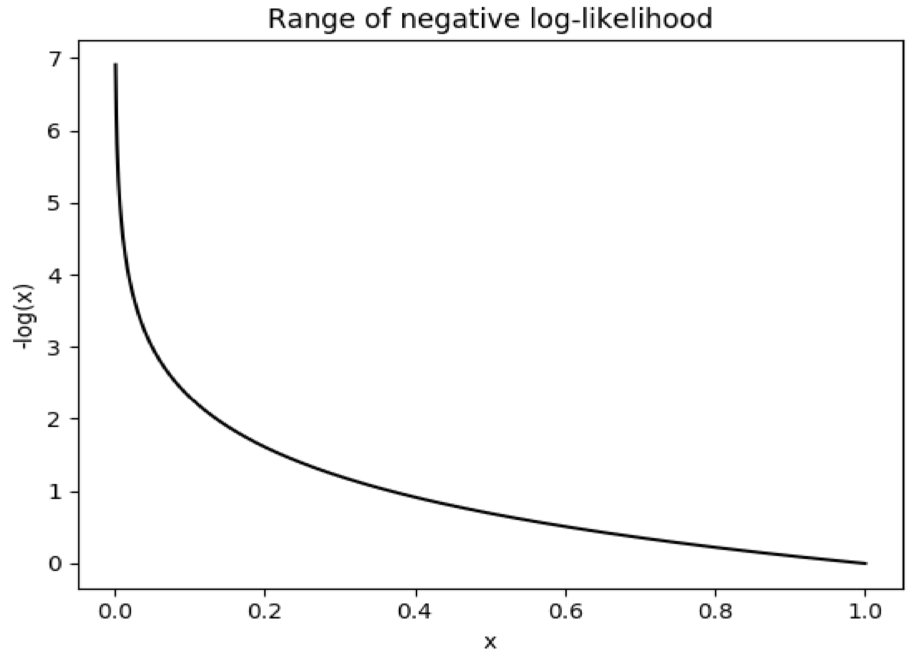
Cross-Entropy Loss

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Multiplying many probabilities/likelihood may lead to very small numbers:
e.g. 0.9*0.1*0.01 = 0.0009 → this is **undesirable**

To avoid this we can express products as **sums** by using the **log function**:

$$\log(a \cdot b) = \log(a) + \log(b)$$



Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat	$\boxed{\mathbf{3.2}}$
car	5.1
frog	-1.7

unnormalized log probabilities

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

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

unnormalized log probabilities

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

	unnormalized log probabilities	$\xrightarrow{\text{exp}}$	unnormalized probabilities	$\xrightarrow{\text{normalize}}$	probabilities
cat	**3.2**		**24.5**		**0.13**
car	5.1		164.0		0.87
frog	-1.7		0.18		0.00

unnormalized log probabilities

probabilities

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

unnormalized probabilities

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

unnormalized log probabilities

probabilities

→ L_i = -log(0.13)
= **2.04**

Softmax Classifier (Multinomial Logistic Regression)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

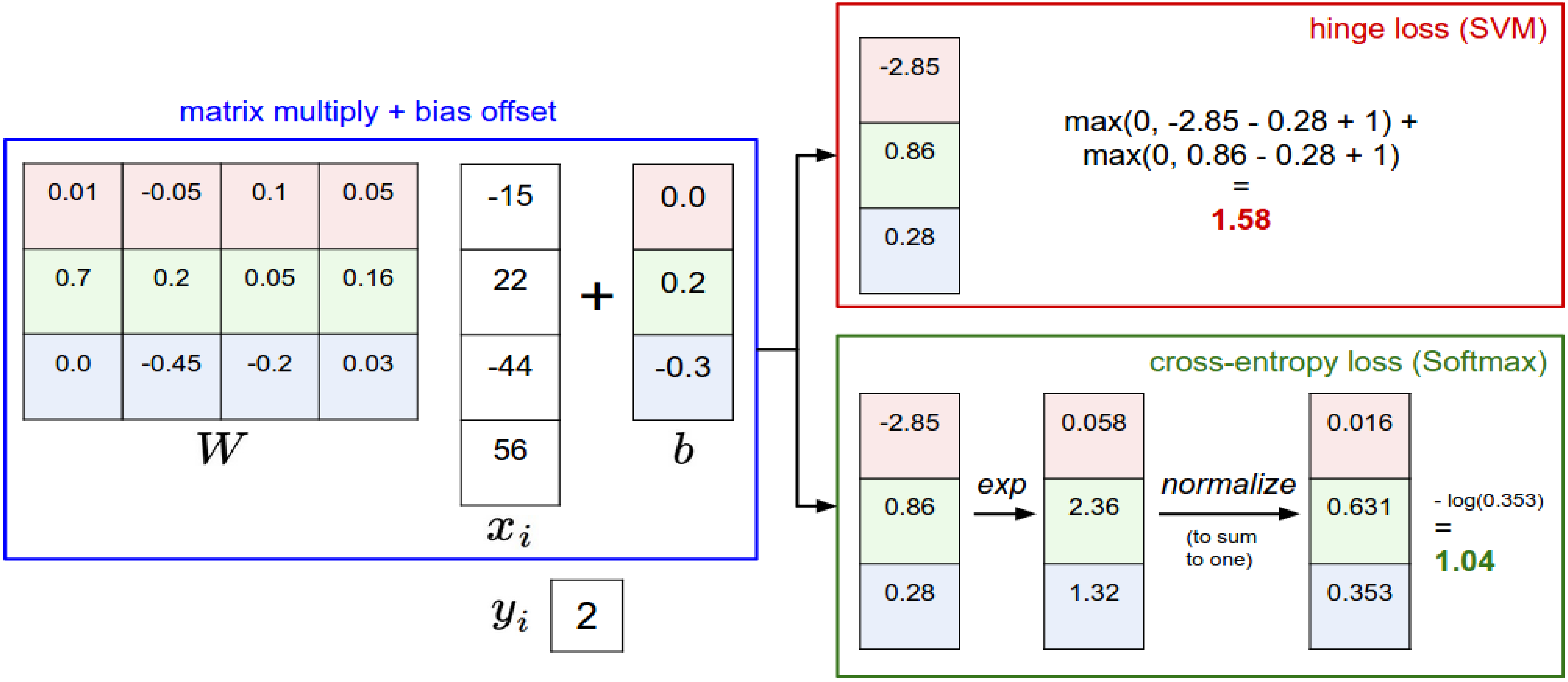
unnormalized probabilities

	unnormalized log probabilities	exp →	unnormalized probabilities	normalize →	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)
= **2.04**

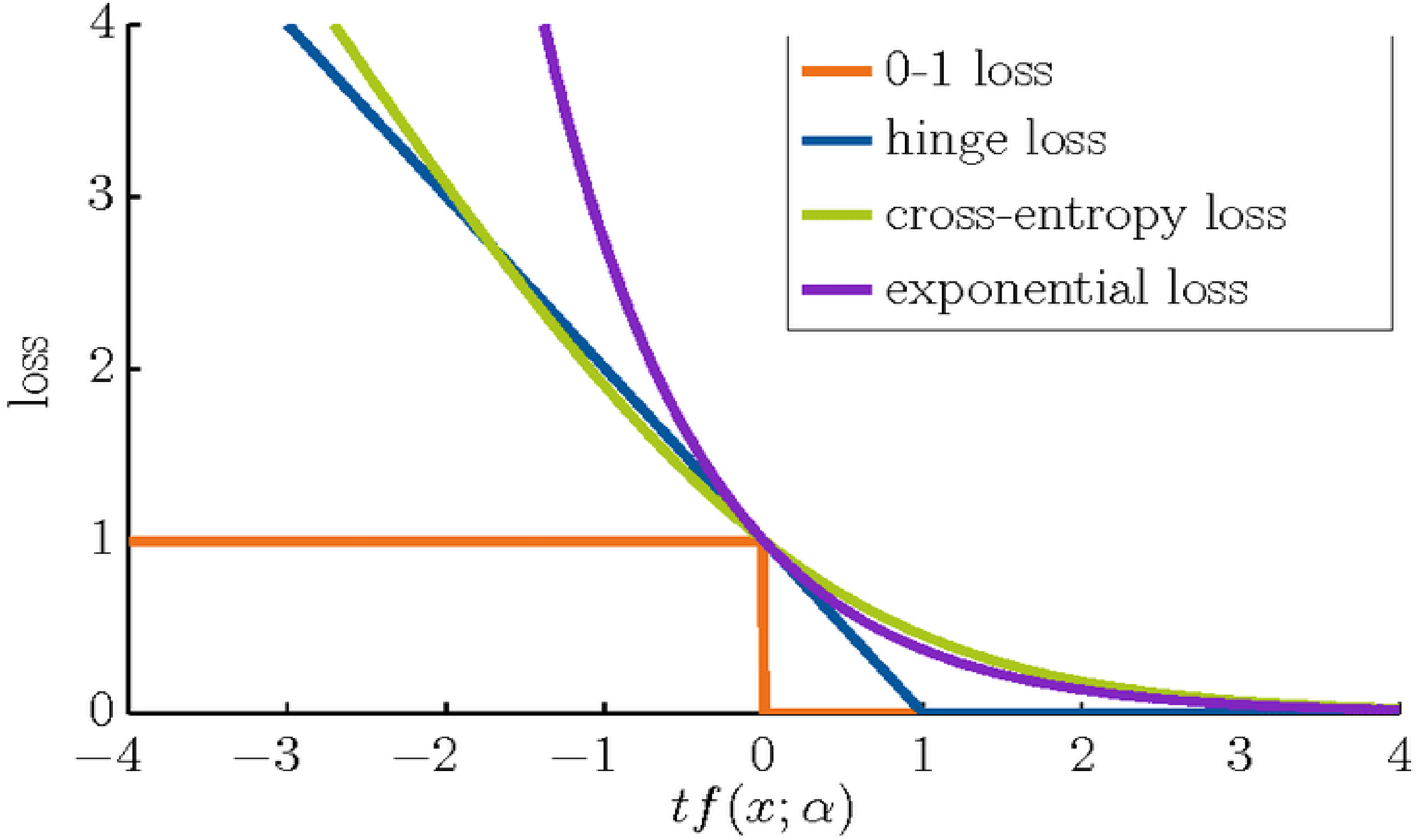
$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

Softmax vs. SVM



Softmax vs. SVM

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$



Summary

- We have some dataset of (x,y)
- We have a **score function:**
- We have a **loss function**:

$$s = f(x; W) \overset{\text{e.g.}}{=} Wx$$

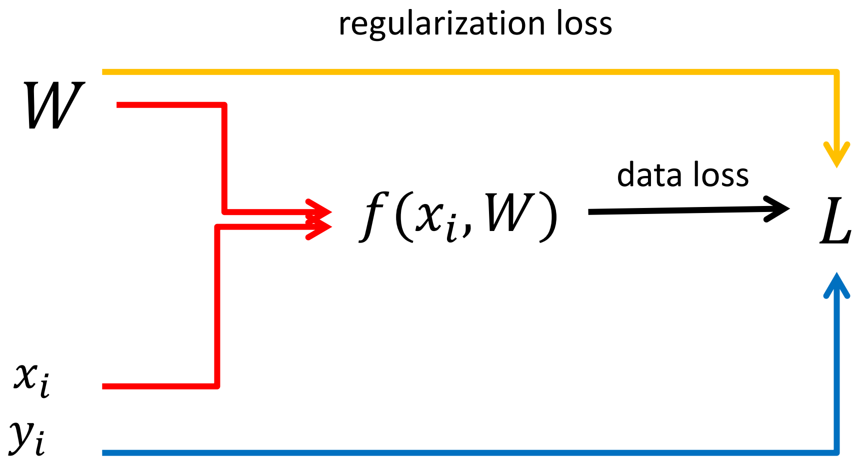
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

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

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$ Complete loss



Summary

- We have some dataset of (x,y)
- We have a **score function:** $s = f(x; W) \overset{\text{e.g.}}{=} Wx$
- We have a **loss function**:

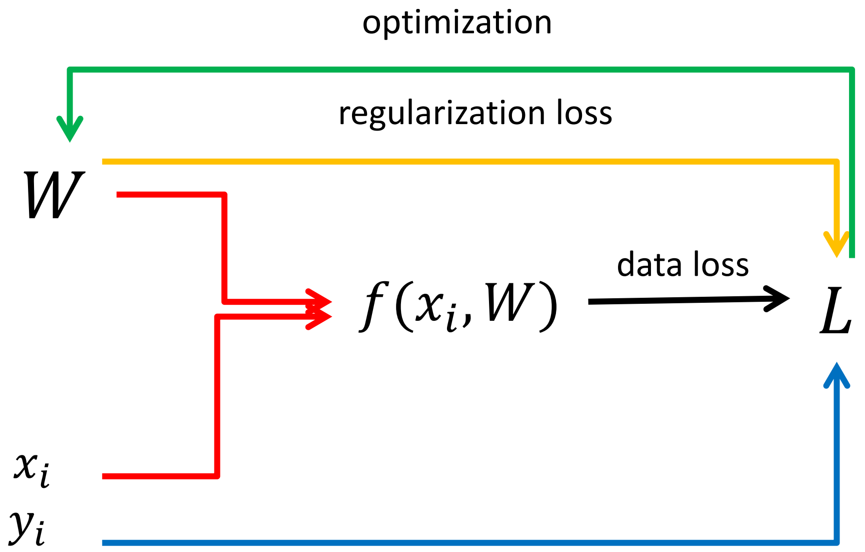
Softmax

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM

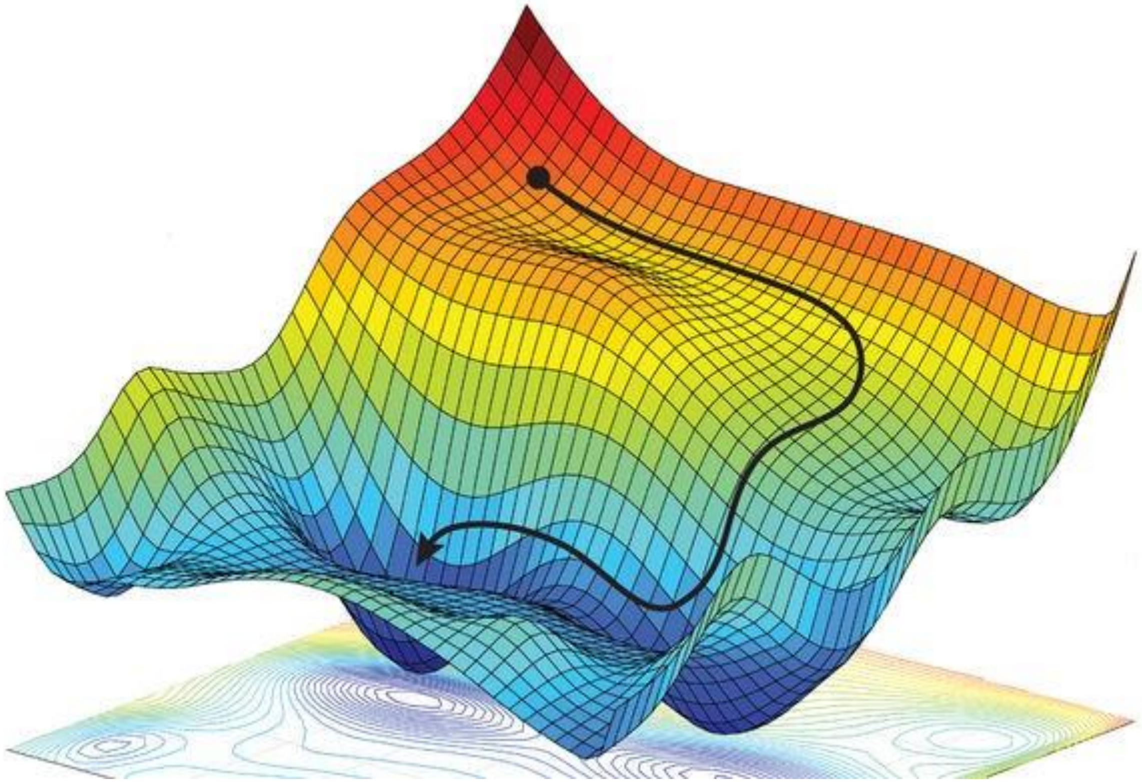
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$$ Complete loss



Optimization

$$w^* = \arg\min_w L(w)$$



Idea: Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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

Idea: Follow the slope

In 1-dimension, the **derivative** of a function gives the slope:

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

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**

Gradient ∇f in 2D

- The gradient of a scalar-valued differentiable function f of several variables, is a vector-valued function $\nabla f : R^n \rightarrow R^n$ whose value at a point is a tangent vector to f.

$$\nabla f = \frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j}$$

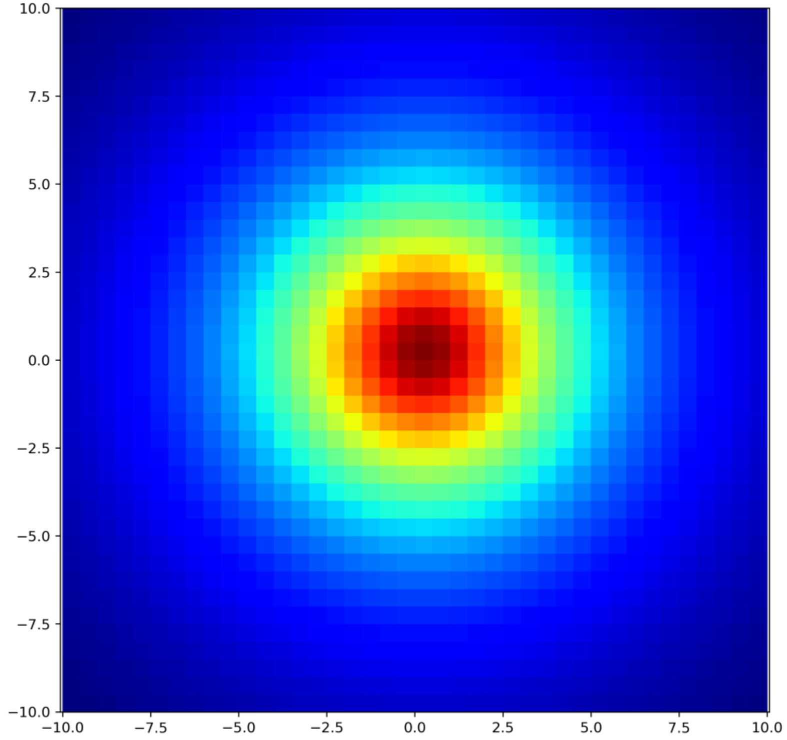
where **i**, **j** are the standard unit vectors in the directions of the *x*, *y* coordinates

Example

```
x = y = np.linspace(-10., 10., 41)
xv, yv = np.meshgrid(x, y, indexing='ij')
fv = h0/(1 + (xv**2+yv**2)/(R**2)) # Some function
```

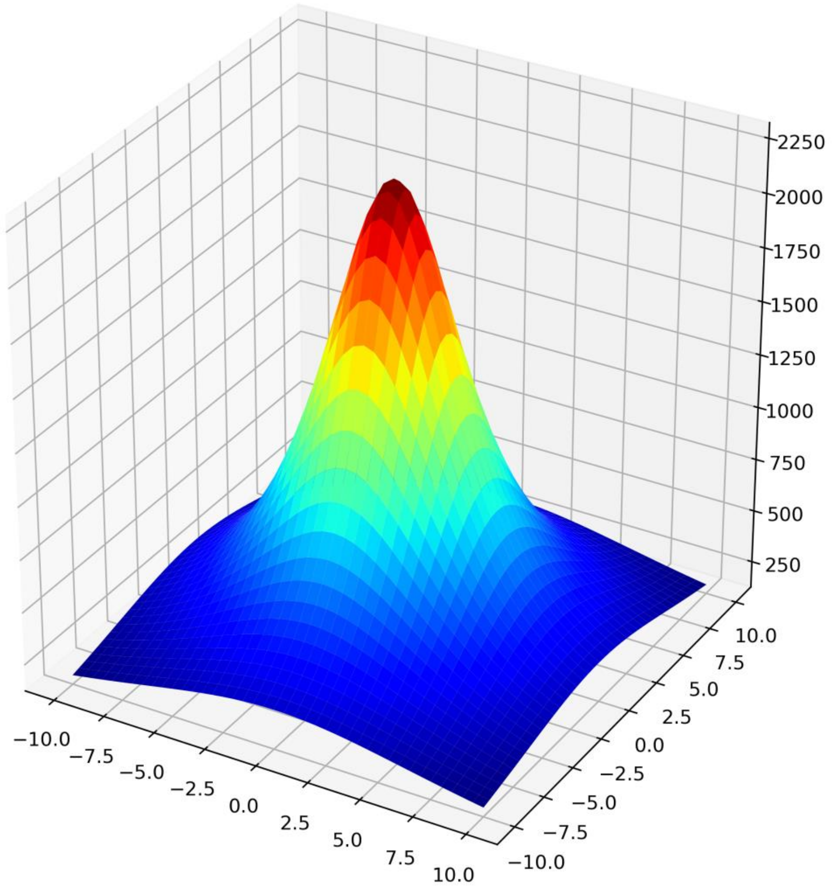
Example

```
plt.pcolormesh(x,y,fv, cmap = 'jet')
```



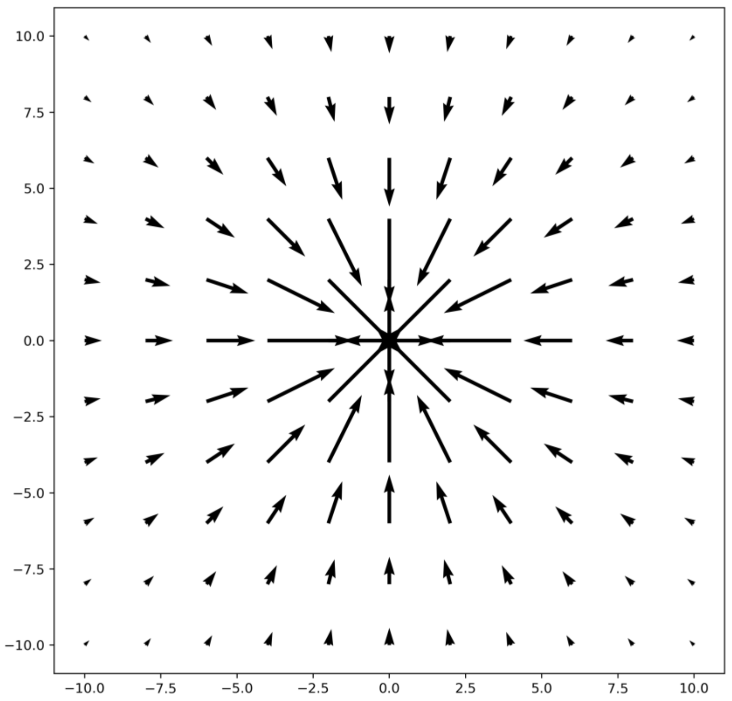
Example

```
ax.plot_surface(xv, yv, fv, cmap='jet')
```

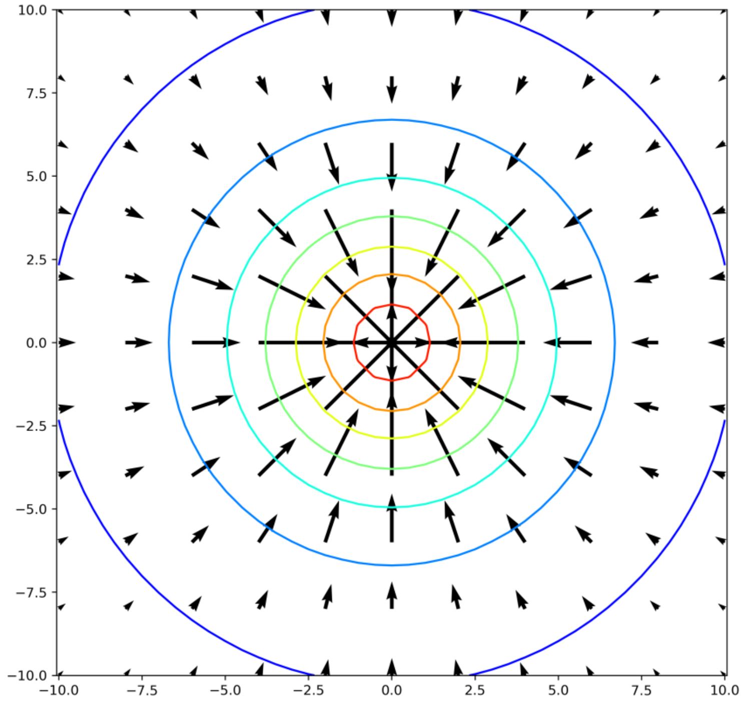


Gradient Computation ∇f

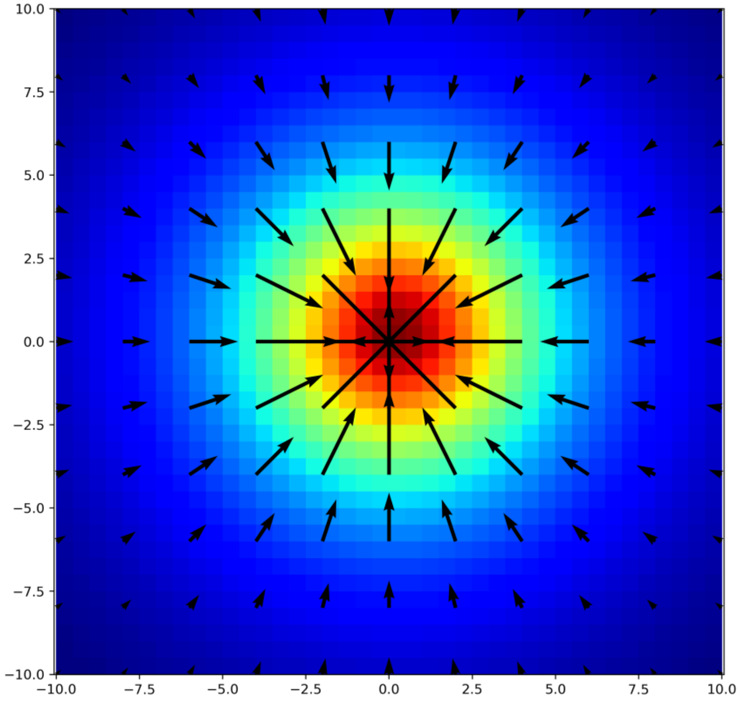
```
dhdx, dhdy = np.gradient(fv) # dh/dx, dh/d
```



gradient



gradient + contour



gradient + function

current W:

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

loss 1.25347

gradient dL/dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[?,
?,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/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 dL/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 dL/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:

[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 dL/dW:

[-2.5,
0.6,
?,
?,
?,
?,
?,
?,
?,…]

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 dL/dW:

[-2.5,
0.6,
0.0,
?,
?,

(1.25347 - 1.25347)/0.0001
= 0.0

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

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 dL/dW:

[-2.5,
0.6,
0.0,
?,
?,

Numeric Gradient:
- Slow: O(#dimensions)
- Approximate

Loss is a function of W: **Analytic Gradient**

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

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

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

want $\nabla_W L$

Use calculus to compute an
analytic gradient

current W:

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

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

gradient dL/dW:

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

current W:

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

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

(In practice we will compute
dL/dW using backpropagation)

gradient dL/dW:

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

Computing Gradients

- **Numeric 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.**

Computing Gradients

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

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

```
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-7):
  """
  sample a few random elements and only return numerical
  in this dimensions.
  """
```

Computing Gradients

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

```
torch.autograd.gradcheck(func, inputs, eps=1e-06, atol=1e-05, rtol=0.001,
raise_exception=True, check_sparse_nnz=False, nondet_tol=0.0)
```

[SOURCE]

Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

The check between numerical and analytical gradients uses `allclose()`.

Computing Gradients

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

```
torch.autograd.gradgradcheck(func, inputs, grad_outputs=None, eps=1e-06, atol=1e-05, rtol=0.001, gen_non_contig_grad_outputs=False, raise_exception=True, nondet_tol=0.0)
```

[SOURCE]

Check gradients of gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` and `grad_outputs` that are of floating point type and with `requires_grad=True`.

This function checks that backpropagating through the gradients computed to the given `grad_outputs` are correct.

Gradient Descent

Iteratively step in the direction of
the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

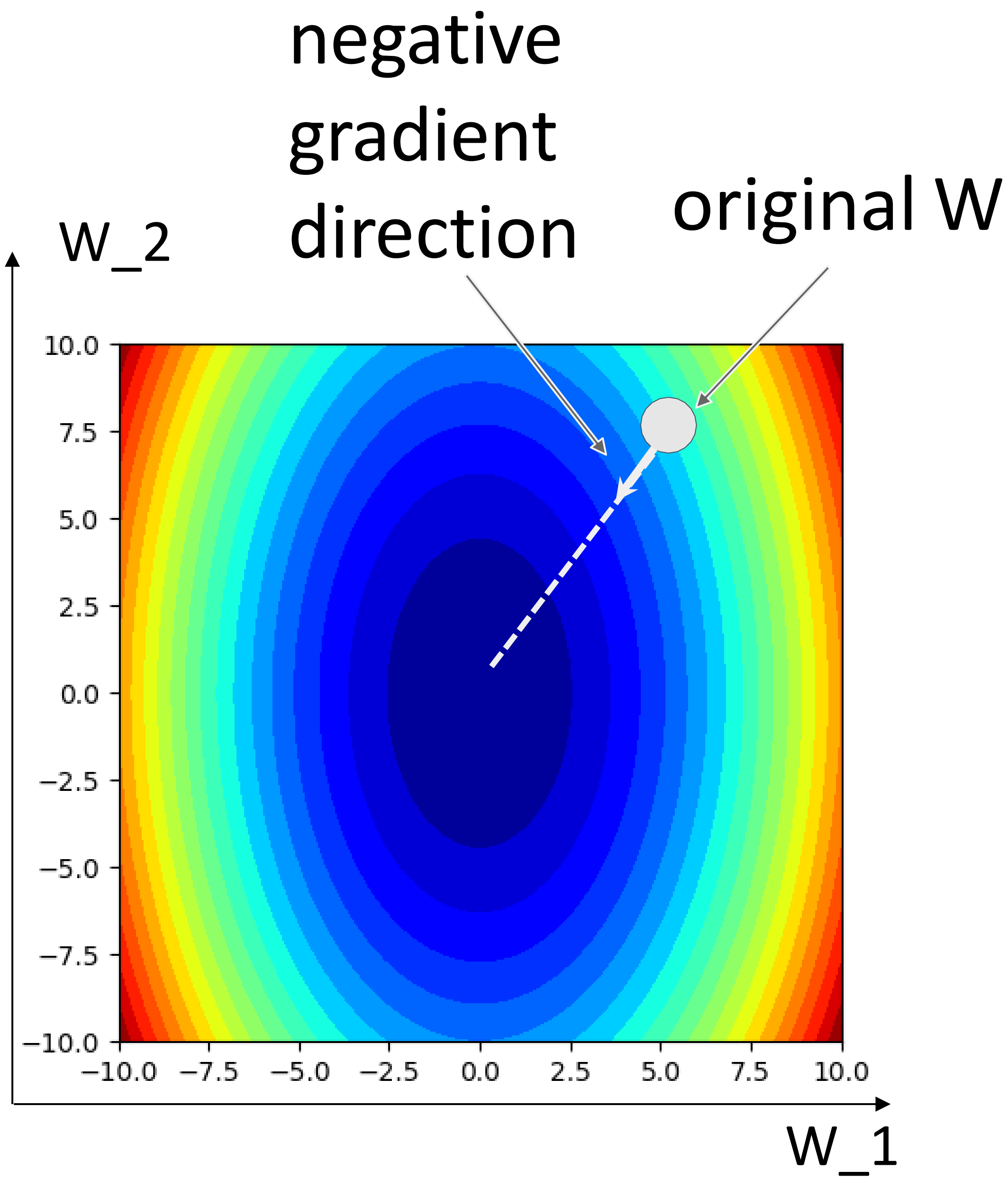
Hyperparameters:
- Weight initialization method
- Number of steps
- Learning rate

Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:
- Weight initialization method
- Number of steps
- Learning rate

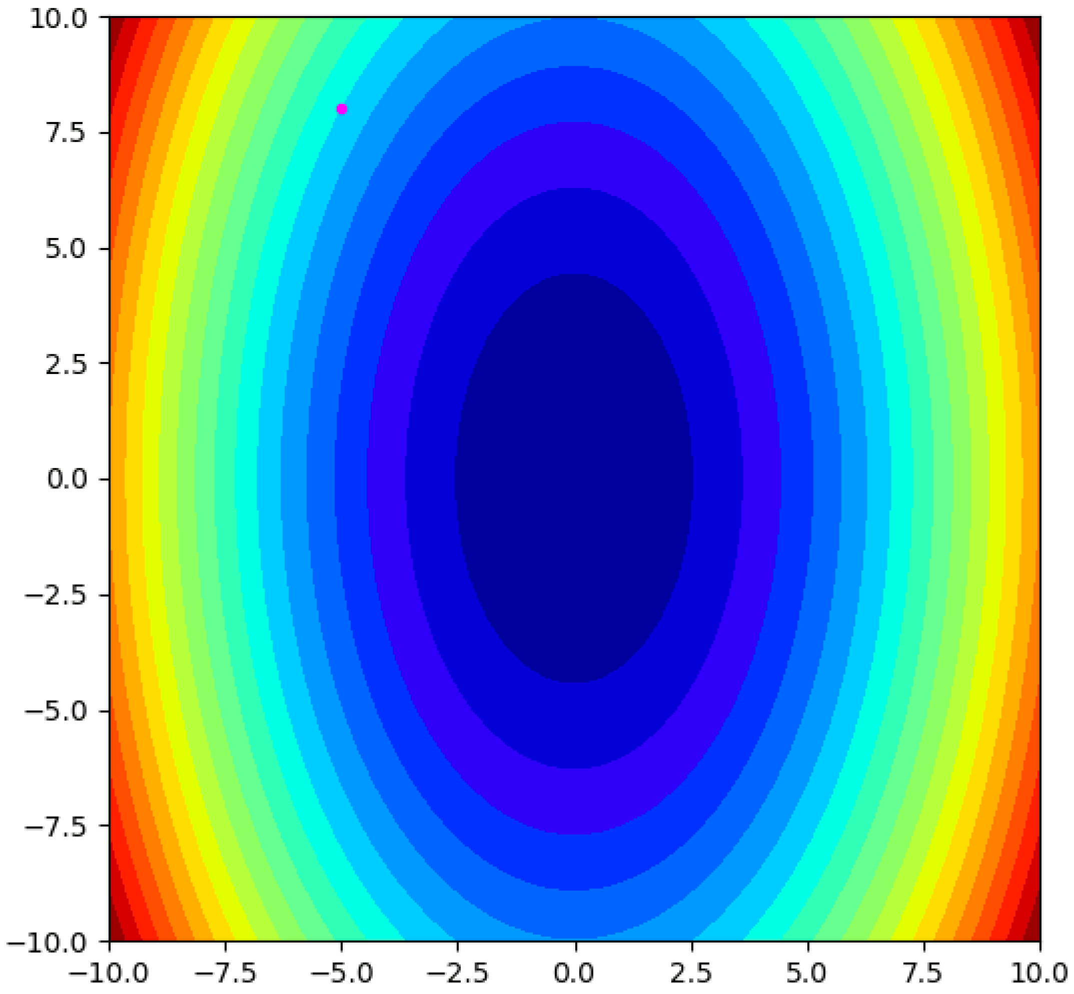


Gradient Descent

Iteratively step in the direction of
the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:
- Weight initialization method
- Number of steps
- Learning rate



Batch Gradient Descent

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

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

Full sum expensive
when N is large!

Stochastic Gradient Descent (SGD)

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

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

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Full sum expensive when N is large!

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

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda R(W)$$

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

Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)]+\lambda R(W)$$

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

Think of loss as an expectation over the full **data distribution** p_{data}

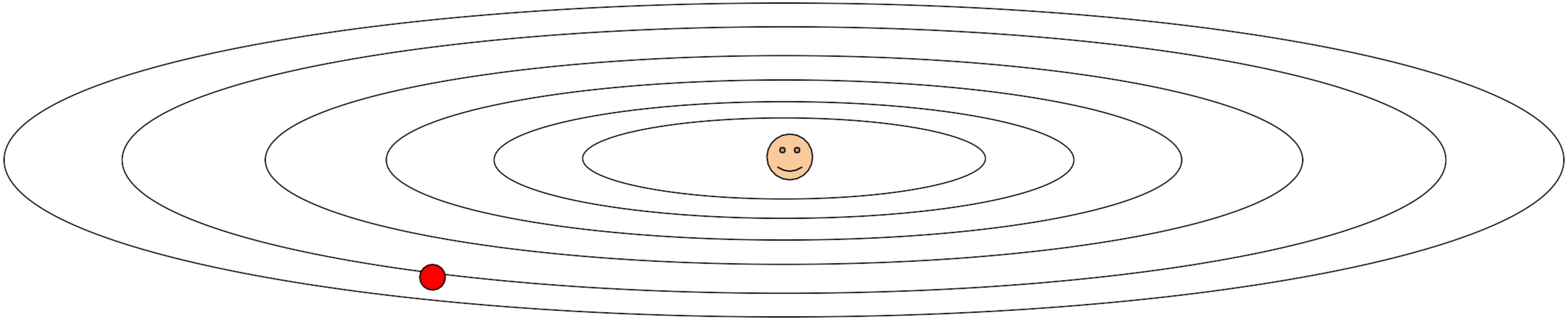
Approximate expectation via sampling

$$\nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda \nabla_W R(W))$$

$$\approx \sum_{i=1}^{N} \nabla_W L_W(x_i, y_i, W) + \nabla_W R(W)$$

Problems with SGD

What if loss changes quickly in one direction and slowly in another?

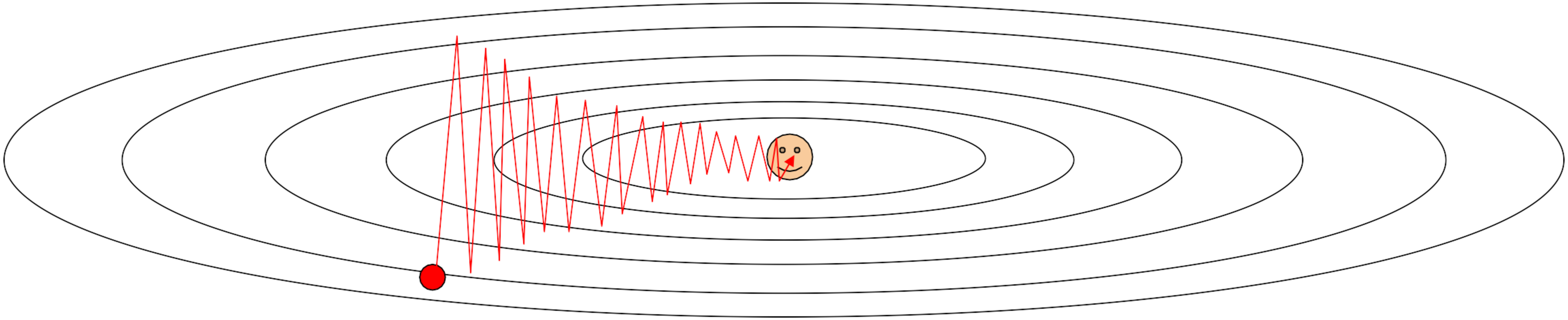


Loss function contour with minimum in the center

Problems with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction



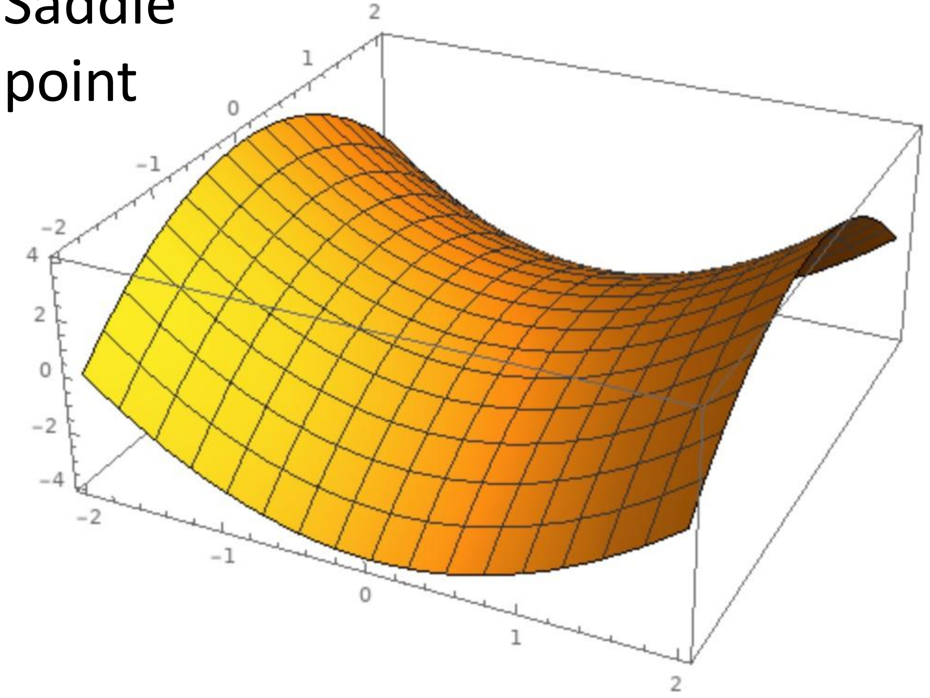
Loss function contour with minimum in the center

Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?

Local Minimum

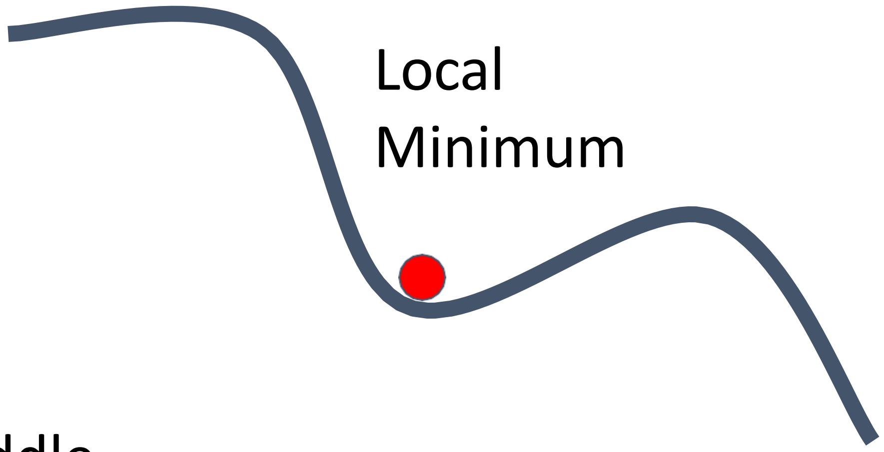
Saddle point

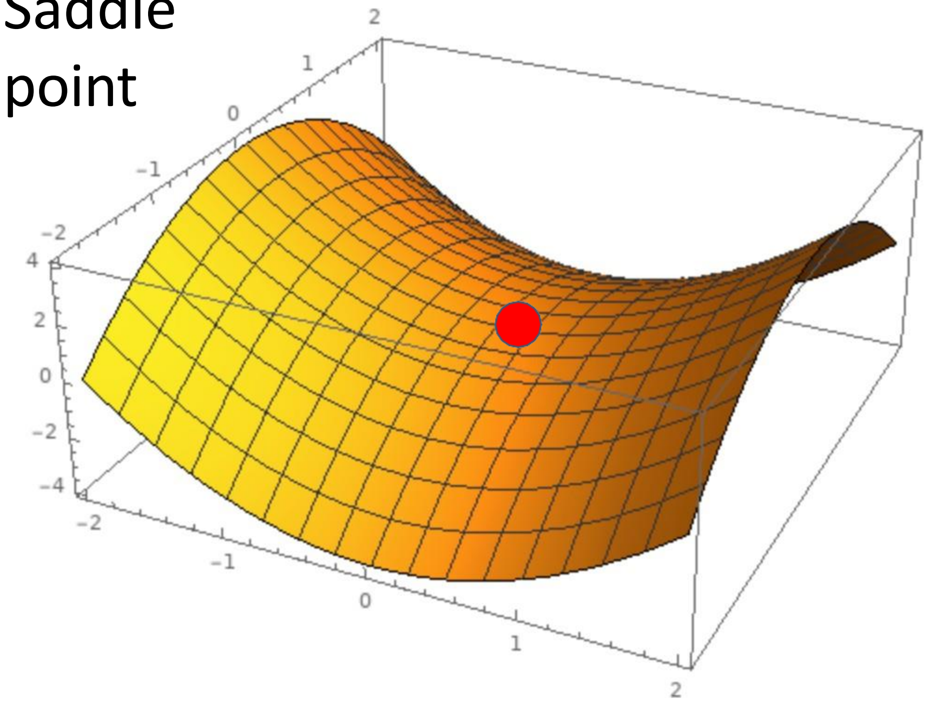


Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?

Zero gradient, gradient descent gets stuck



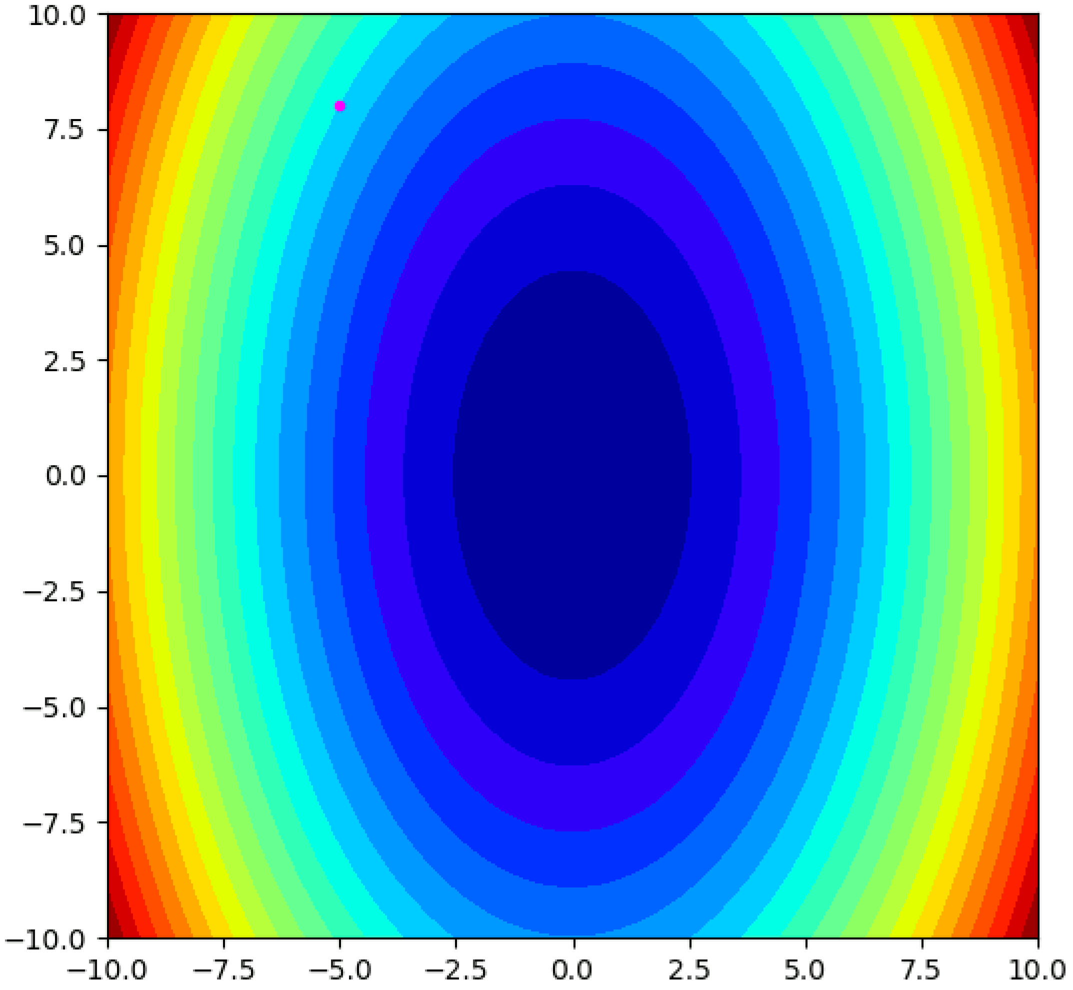


Problems with SGD

Gradients are calculated from minibatches → they can be **noisy**

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

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



SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

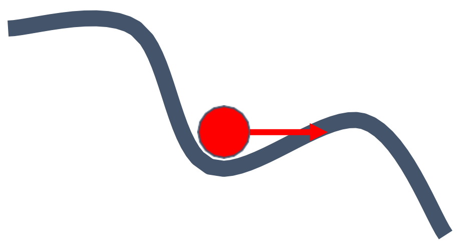
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

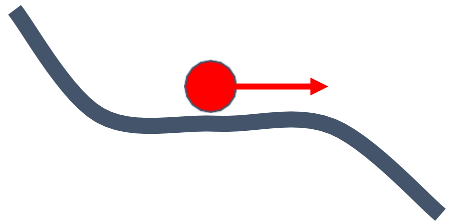
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

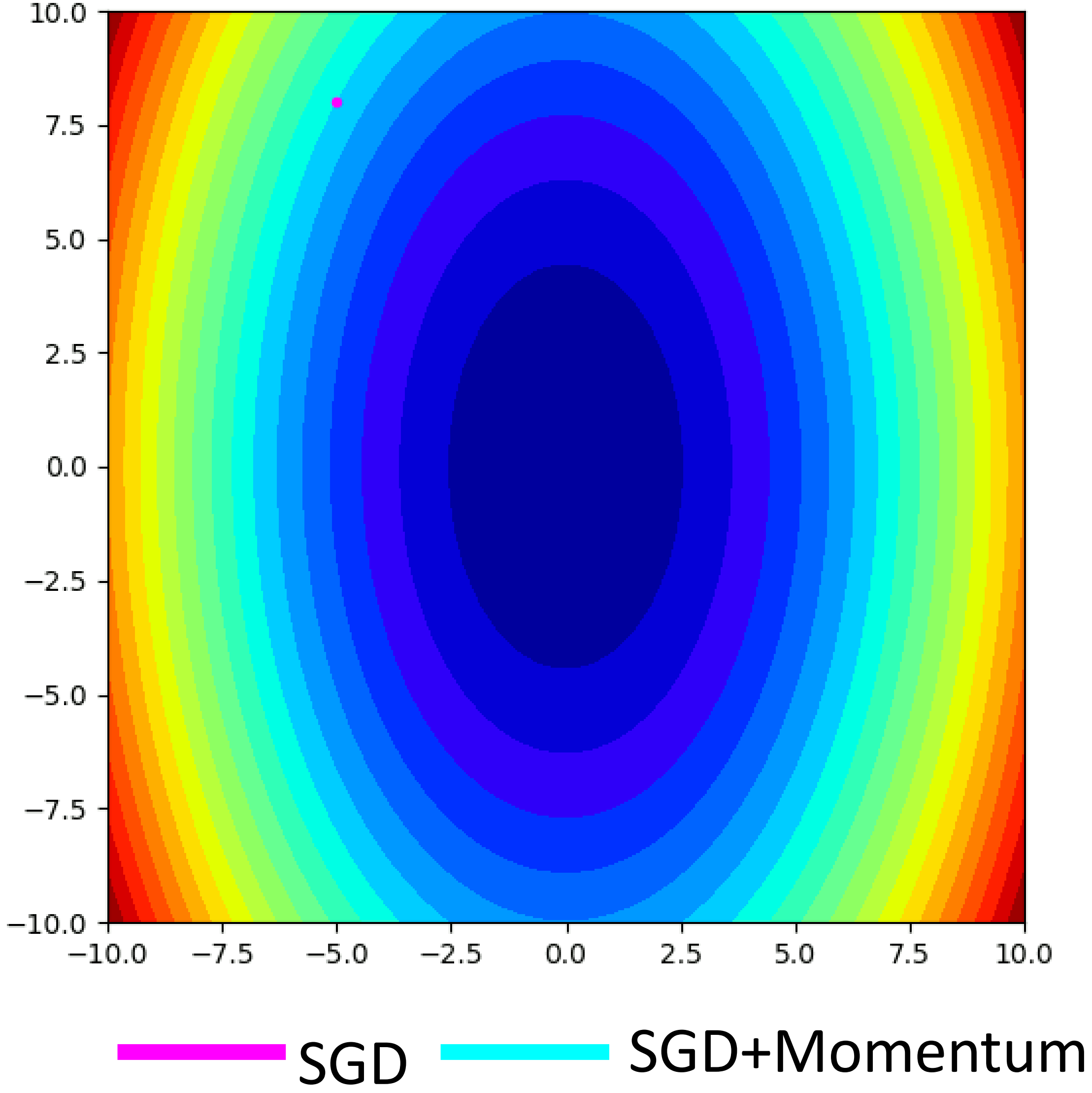
Local Minima



Saddle points



Gradient Noise



AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

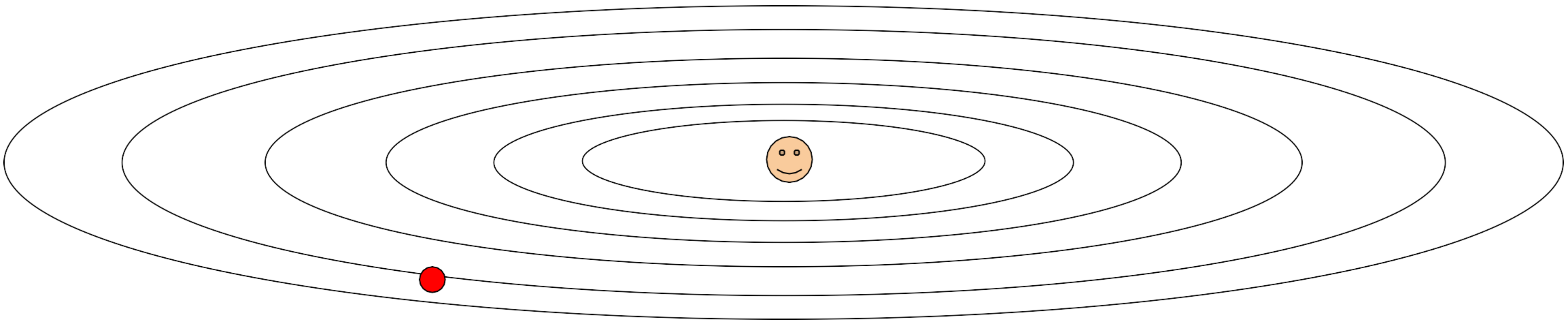
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates"
or "adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



Progress along “steep” directions is damped; progress along “flat” directions is accelerated

RMSProp

```python
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

AdaGrad

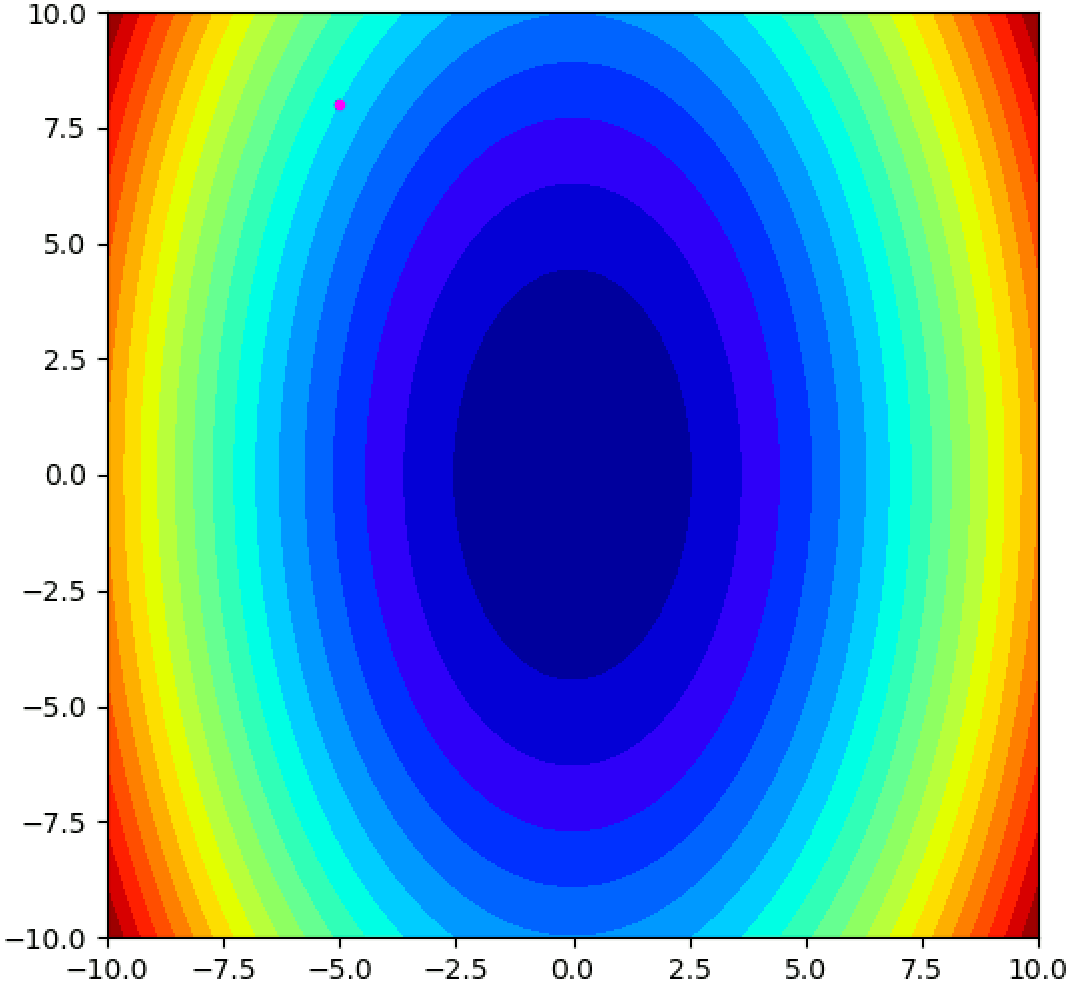
↓

```python
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Tieleman and Hinton, 2012

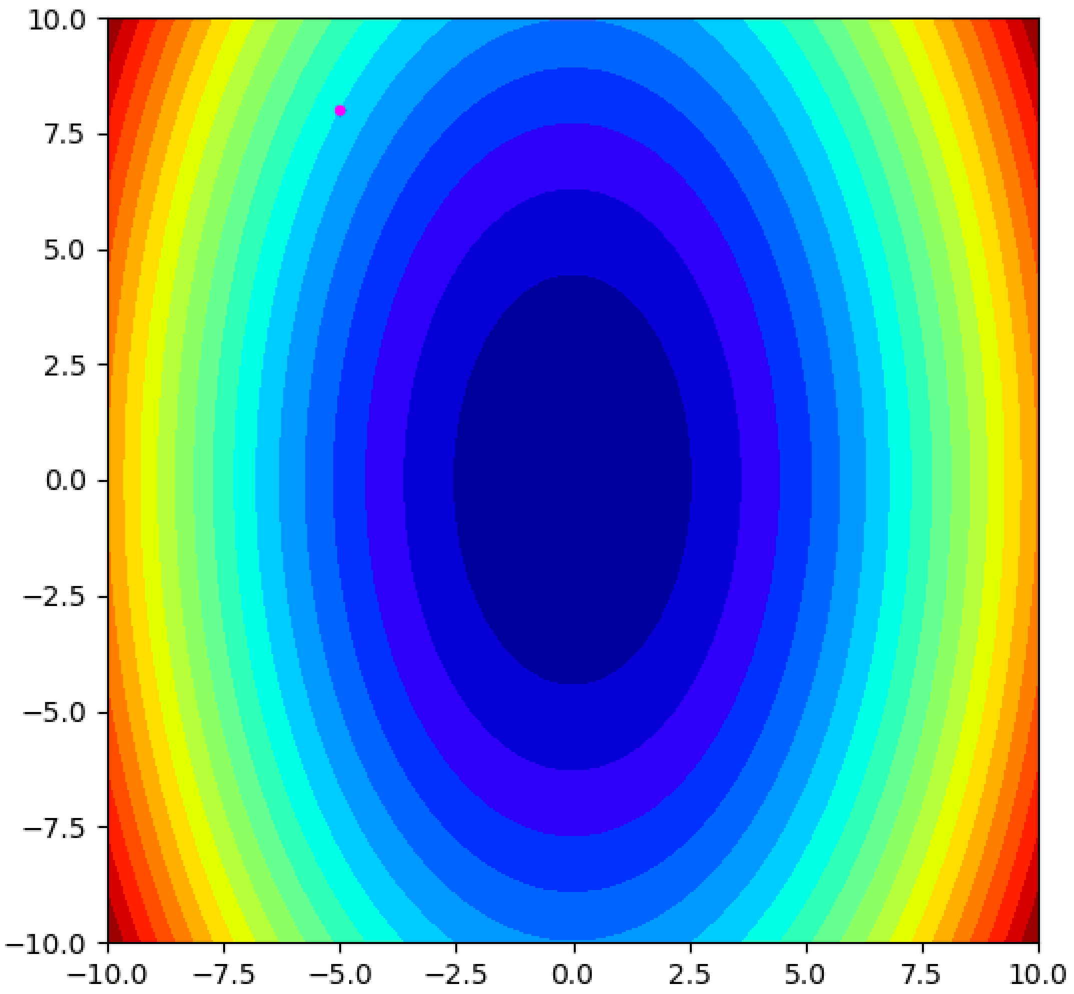
RMSProp

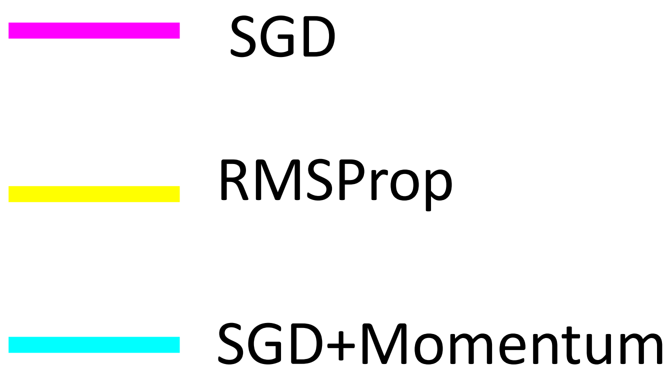


SGD

RMSProp

RMSProp Noise





Adam: RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam: RMSProp + Momentum

```python
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

```python
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

SGD+Momentum

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam: RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam: RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Momentum

AdaGrad / RMSProp

Bias correction

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam: RMSProp + Momentum

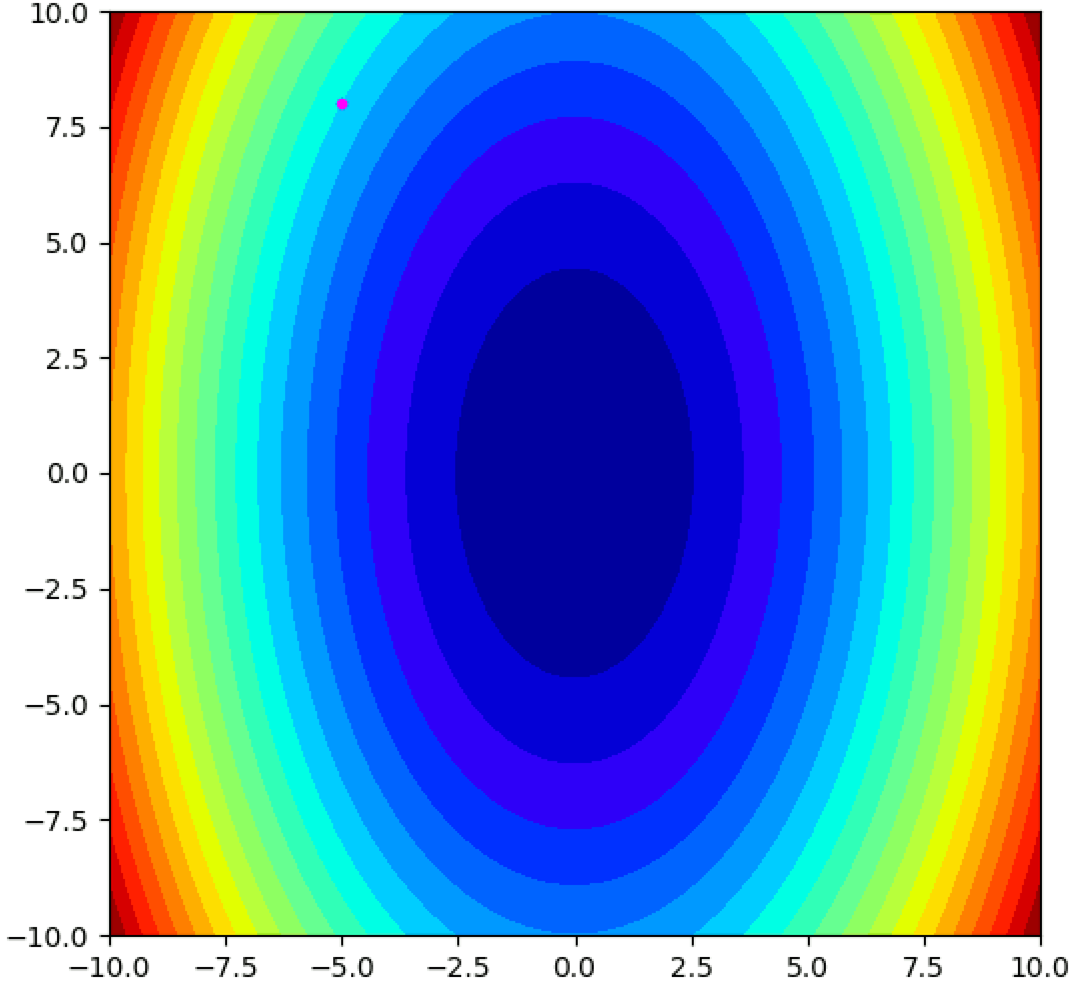
```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Adam example values: beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam



SGD

SGD+Momentum

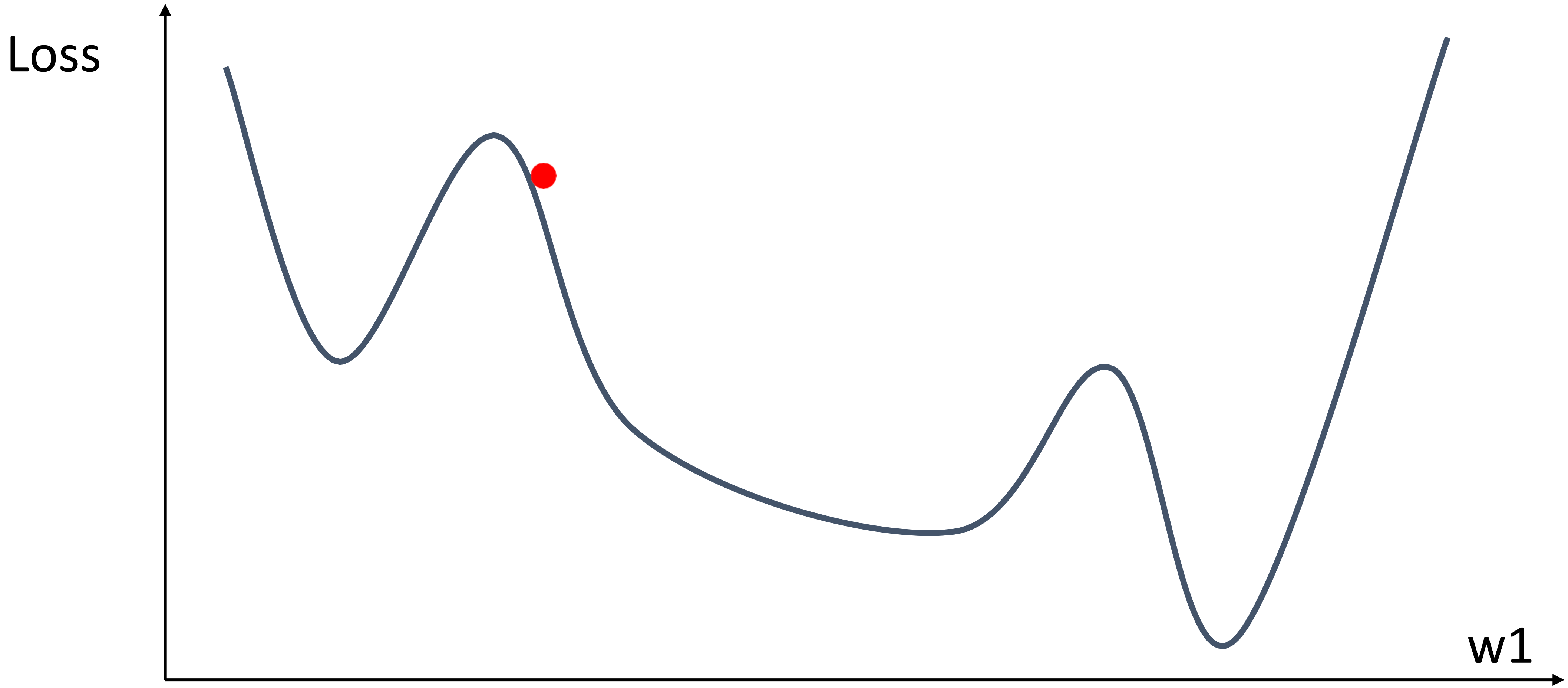
RMSProp

Adam

Optimization Algorithm Comparison

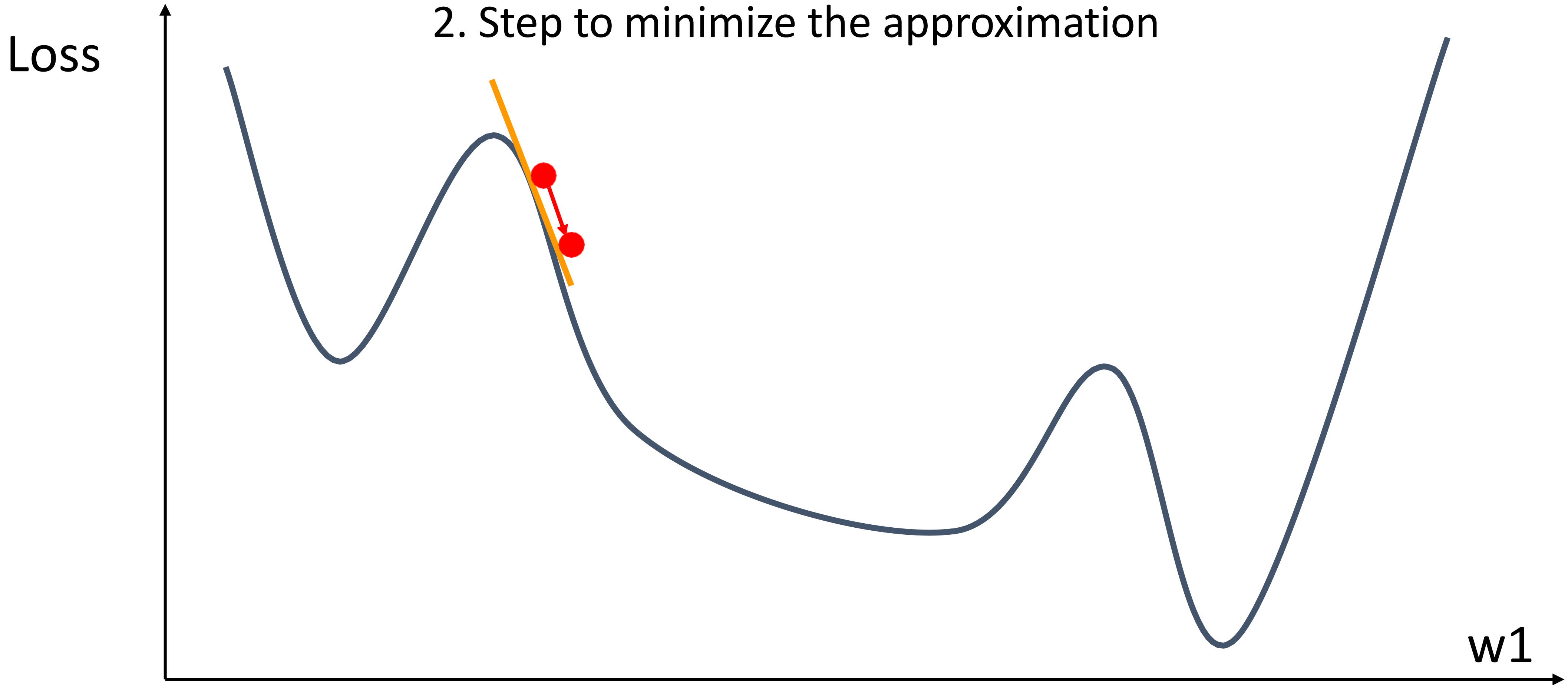
Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

So far: First-Order Optimization



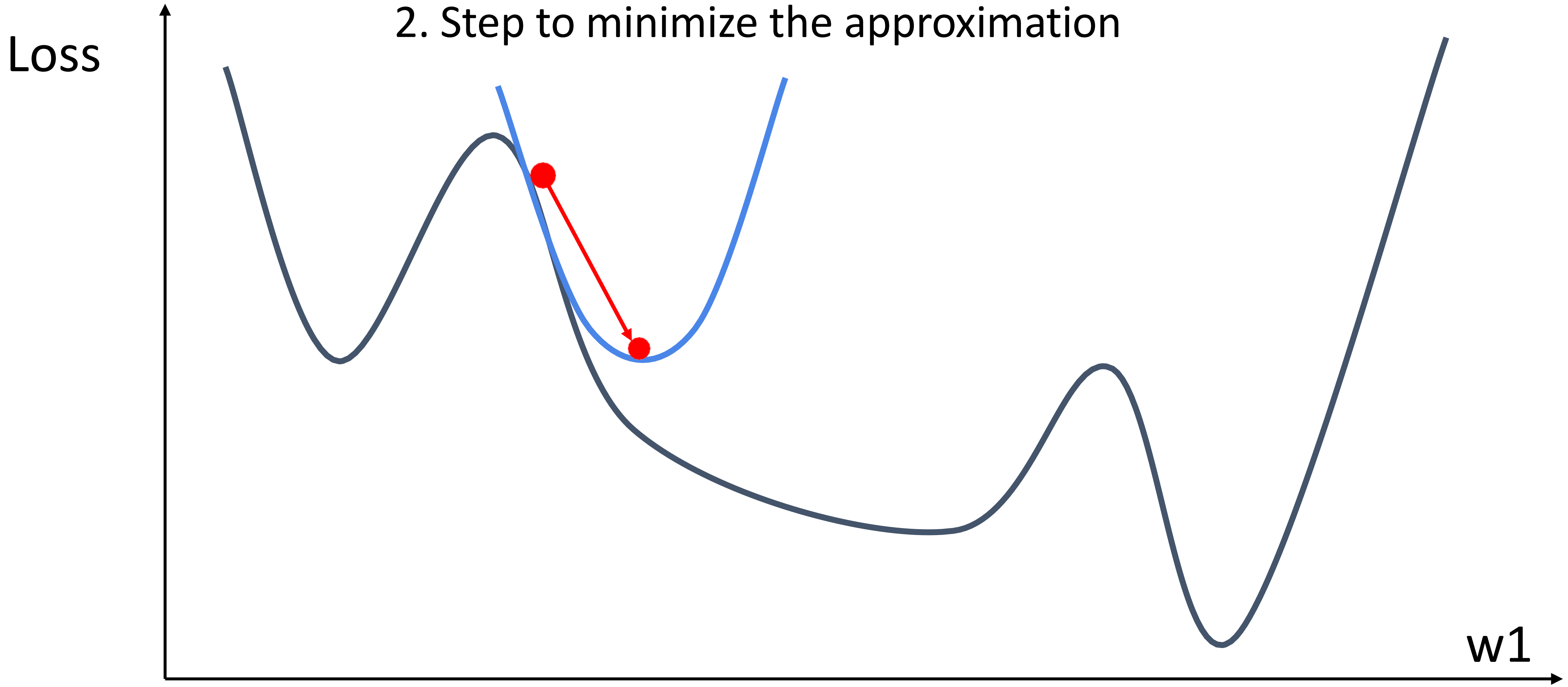
So far: First-Order Optimization

1. Use gradient to make linear approximation
2. Step to minimize the approximation



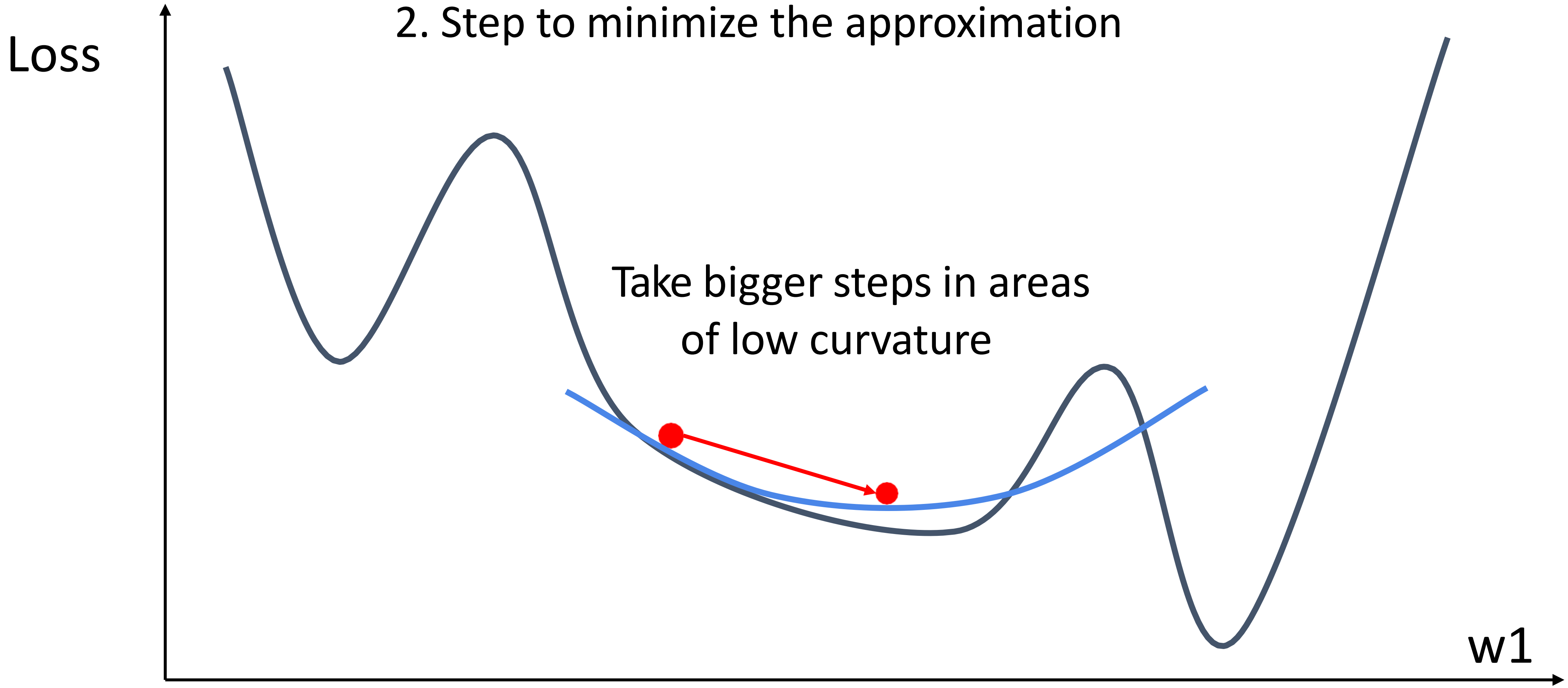
Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Second-Order Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Second-Order Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

In practice:

- **Adam** is a good default choice in many cases
  SGD+Momentum can outperform Adam but may require more tuning

- If you can afford to do full batch updates then try out **L-BFGS** (and don’t forget to disable all sources of noise)

Summary

1. Use **Linear Models** for image classification problems

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

2. Use **Loss Functions** to express preferences over different selection of weights

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$ Softmax

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

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

3. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```