Lecture 4 - Regularization + Optimization

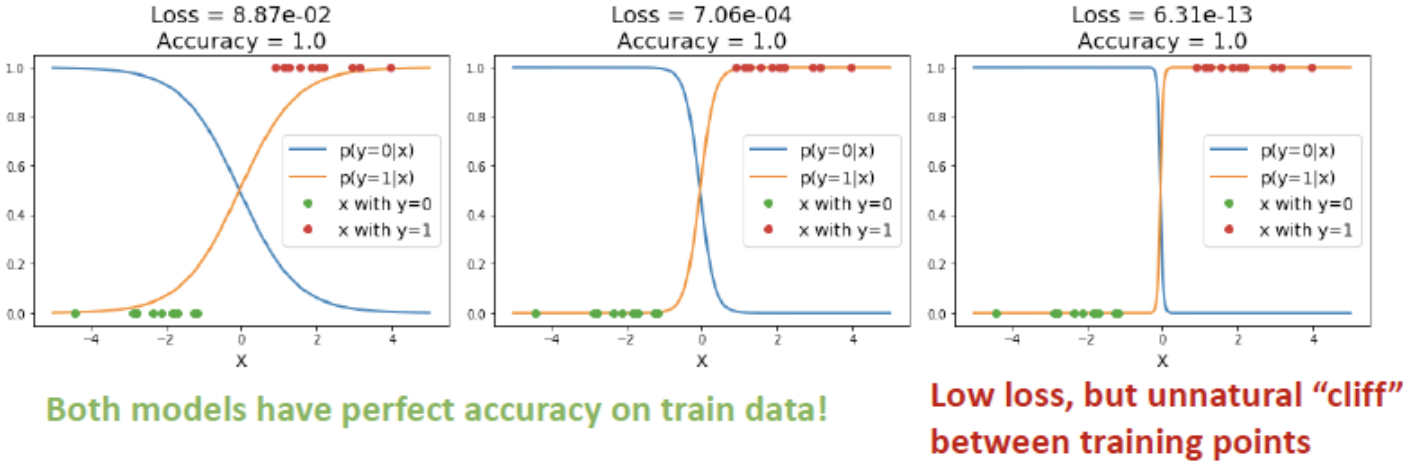
1. Regularization

- **Overfitting**

A model is **overfit** when it performs too well on the training data, and has poor performance for unseen data

Eg. Linear classifier with 1D inputs, 2 classes, softmax loss

$$s_i = w_i x + b_i$$
$$p_i = \frac{\exp(s_i)}{\exp(s_1) + \exp(s_2)}$$
$$L = -\log(p_y)$$



--> p_y: the predicted probability of true categories

green and red dots: training data with labels

blue and orange line: probability predictions based on the data

from left graph to right graph: larger weights + lower loss --> however, the cliff between training points [regions with no training data] (more complicated model --> small change in data will result in large change in prediction)

- **Regularization**

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\lambda R(W)}_{\text{Regularization: Prevent the model from doing } \textit{too} \text{ well on training data}}$$

λ is a hyperparameter giving regularization strength

, where $\lambda > 0$ --> to penalize model complexity and thre regularization part does not depend on any training data

Loss function consists of **data loss** to fit the training data and **regularization** to prevent overfitting

- $R(W)$ Examples:
  - Simple: L2 regularization: $R(W) = \sum_{k,l} W_{k,l}^2$
    L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$
  - Complex: Dropout, batch normalization, cutout, mixup, stochastic depth, etc...
- --> prefer simpler models (generally perform better on unseen data)

  [Regularization term causes loss to increase for model with sharp cliff]
-

L2 Regularization

$$R(W) = \sum_{k,l} W_{k,l}^2$$

$x = [1, 1, 1, 1]$

$w_1 = [1, 0, 0, 0]$

$w_2 = [0.25, 0.25, 0.25, 0.25]$

L2 regularization prefers weights to be "spread out"

more robust to noise

$w_2^T x < w_1^T x$

smaller

regularization

Predicted Output:

$w_1^T x = w_2^T x = 1$

Same predictions, so data loss will always be the same

2. Optimization

--> find the optimal weight to minimize the loss function $W^* = argmin_w L(w)$

- **Computing Gradients**
  - Numeric gradient: approaximate, slow, easy to write --> follow the slope

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

    - In multiple dimensions, the gradient is the vector of (partial derivatives) along each dimension
    - Slope in any direction: dot product of the direction with the gradient (direction is **negative**!)
  - Analytic gradient: exact, fast, error prone (hard to calculate)

    --> Use calculus to compute an analytic gradient

  In practice we will compute dL/dW using **backpropagation**

  - Gradient Check: Always use analytic gradient for **implementation**, but **check** implementation with numerical gradient

- **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 / iterations
    - Learning rate
  - Batch Gradient Descent --> GD for the entire dataset

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

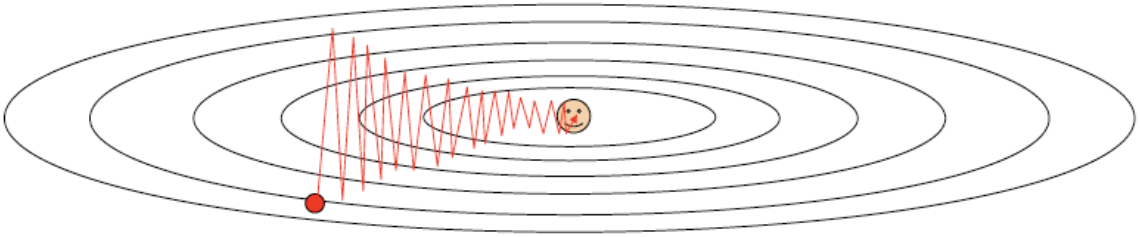
Full sum expensive when N is large!

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

  - Stochastic Gradient Descent (SGD) --> GD for a **minibatch** of examples (commonly 32 / 64 / 128 data points)
    - Bigger the batch is , better the optimization will be, however with more GPU and memory usage
    -

```
# 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
```

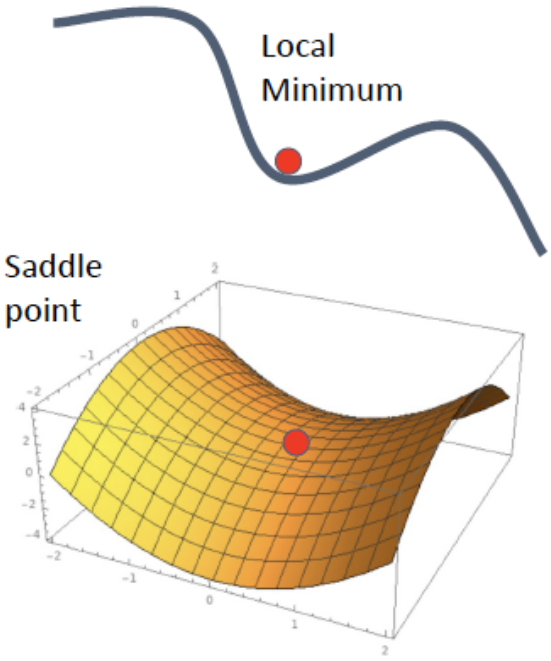
- Hyperparameters:
  - Weight initialization method
  - Number of steps / iterations
  - Learning rate
  - Batch size
  - Data sampling
- Problems with SGD
  - Loss changes quickly in vertical direction but slowly in horizontal direction --> Very slow progress along shallow dimension, jitter along steep direction



  - Local minimum / saddle point

    --> gradient becomes zero and get stuck

    --> BGD must get stuck while SGD may or may not get stuck, depending on the data sampling



    - Gradients come from minibatches so they can be **noisy**!

- **Optimization Techniques**
    - SGD

      Note: x_t is the weight to be optimized

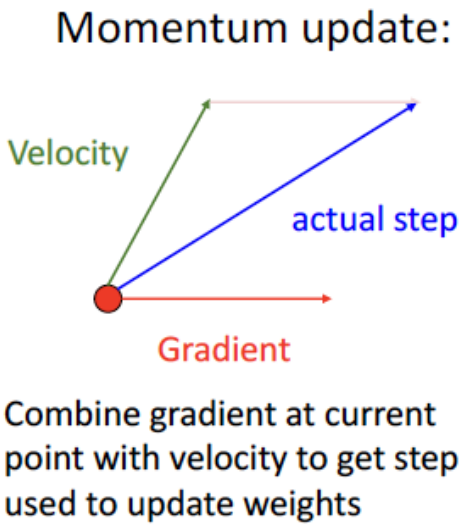
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

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

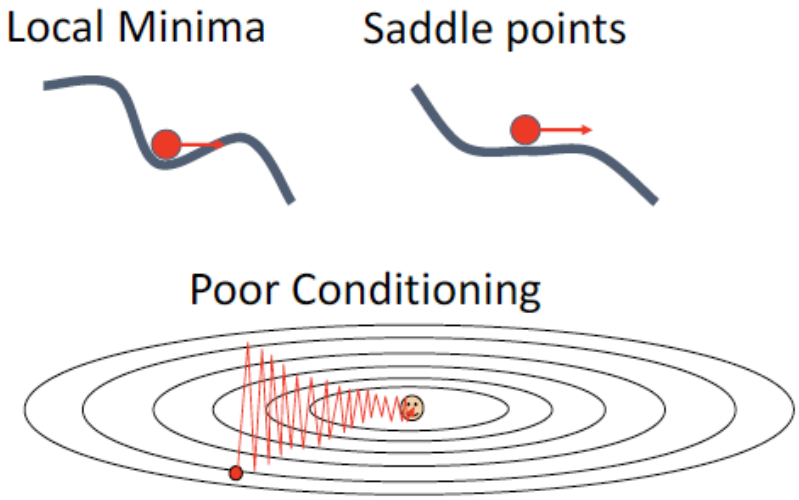
    - SGD + Momentum

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

```
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 --> weight converges more and more faster
- Rho gives “**friction**”; typically rho=0.9 or 0.99
- Feature: fast but may overshoot the global value



- 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)
```

- Add **element-wise scaling** of the gradient based on the historical sum of squares in each dimension (1e-7 is used to avoid denominator to be 0)
- “Per-parameter learning rates” or “adaptive learning rates” --> updated during optimization
- Feature:

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

--> problem: if the training process takes long time, *grad_squared* will become larger and larger --> the process may stop before even reaching the minimum

- RMSProp: "Leaky Adagrad"

```
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)
```

--> Adagrad leaks over time

- 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 (as assume `beta2 = 0.999`, then `moment2` is very small at the first few iterations --> may have unstable steps for the first few iterations)

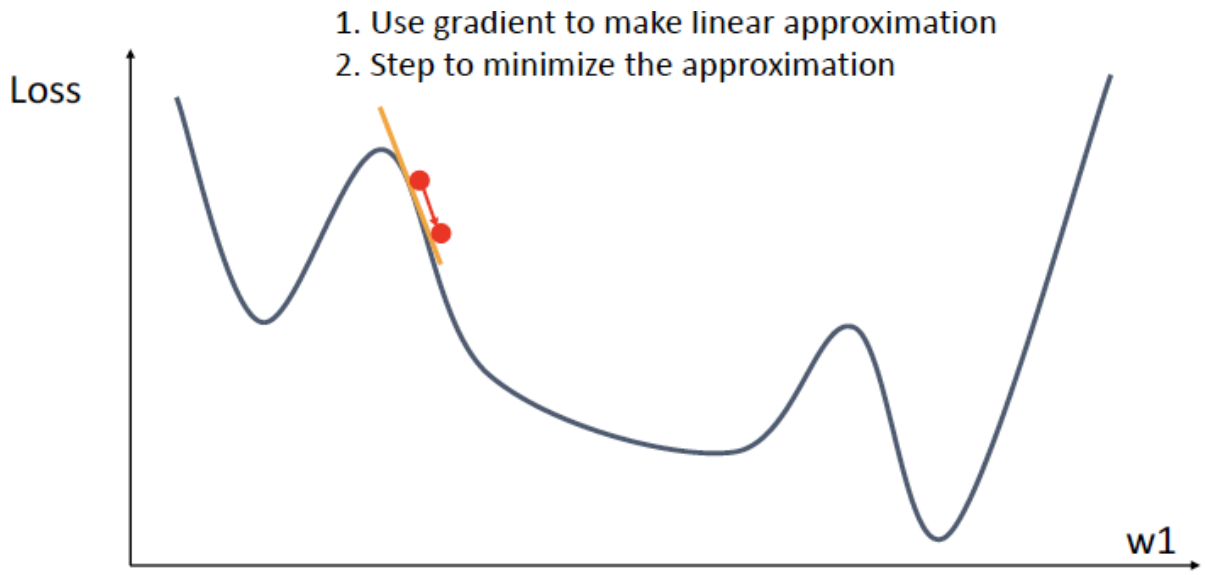
  - Adam with **beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4** is a great starting point for many models!

- Summary

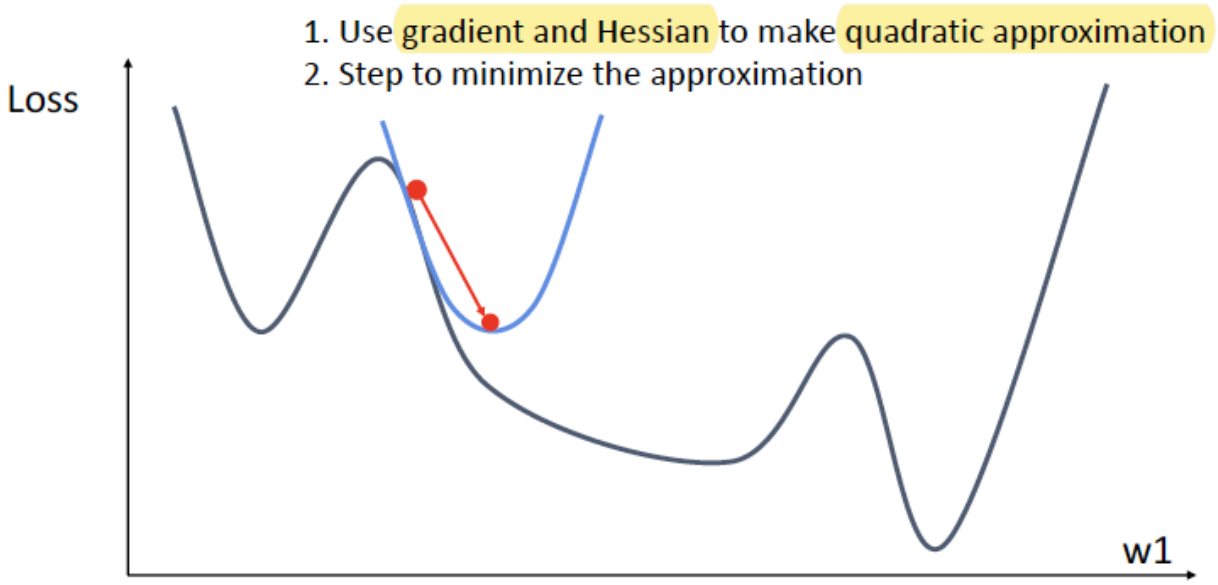
Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
Nesterov	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

AdamW(Adam with decoupled weight decay) should probably be your “default” optimizer for new problems (use the same parameters as Adam)

- **First and Second-Order Optimization**
  - First-order Optimization



- Second-order Optimization



--> can take bigger steps especially in areas of low curvature (make larger changes of gradients)

  - Impractical:
    - Hessian has O(N^2) elements;
    - Inverting takes O(N^3);
    - The number of learnable parameters N = (Tens or Hundreds of) Millions