LECTURE 04

Training Neural Networks (part 2)

Classification, Stochastic Gradient Descent, Loss Surfaces, Convergence Issues, Momentum

CS/DS 541: Deep Learning, Fall 2025 @ WPI

Fabricio Murai

Where are we?

- Learning is different than optimization
- Feed-forward neural networks combine linear layers with non-linear activation functions
  - We need hidden layers + activation functions to learn models that work for non-linearly separable data
- Neural Nets (NNs) are universal approximators
- NNs are most often trained with gradient based methods
  - We need a loss function to compute its gradient with respect to parameters
  - Gradient Descent moves params toward opposite direction of gradient

Classification
(we will blaze through Logistic Regression)

Binary classification

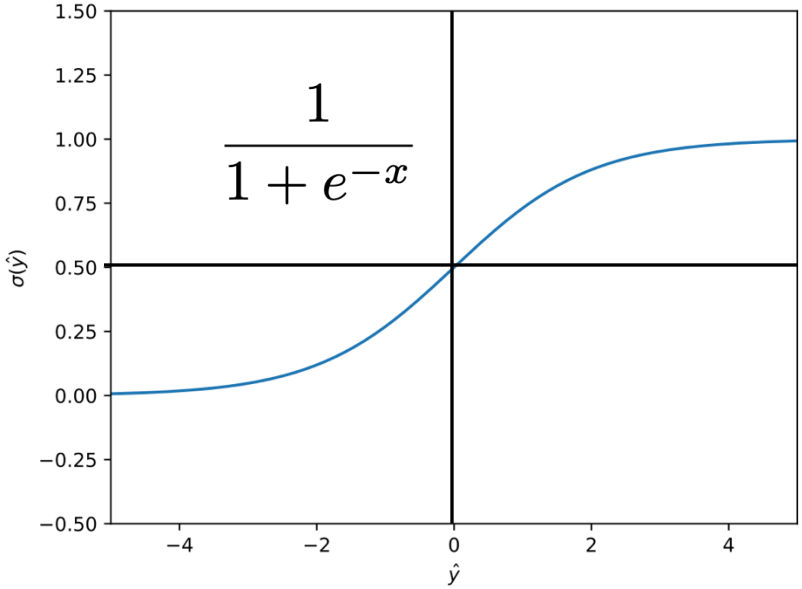
- Simplest classification problem: 2 classes, i.e., $y \in \{ 0, 1 \}$.

Classifier → Chihuahua or Blueberry Muffin

- One of the simplest and most common classification techniques is **logistic regression**.
- Logistic regression is similar to linear regression but applies a sigmoidal "squashing" function to ensure that $\hat{y} \in (0, 1)$.
  - Sigmoid is an example of *activation function*

Sigmoid: a "squashing" function

- A sigmoid function is an "s"-shaped, monotonically increasing and bounded in (0, 1).
  - Good for representing probabilities
- Crosses y-axis at y = 0.5
  - "Smooth" transition
- Derivative:
  - Largest at x = 0
  - Very small for $x < -4$ or $x > 4$



Logistic Regression

- Model computes “logits" similar to linear regression:

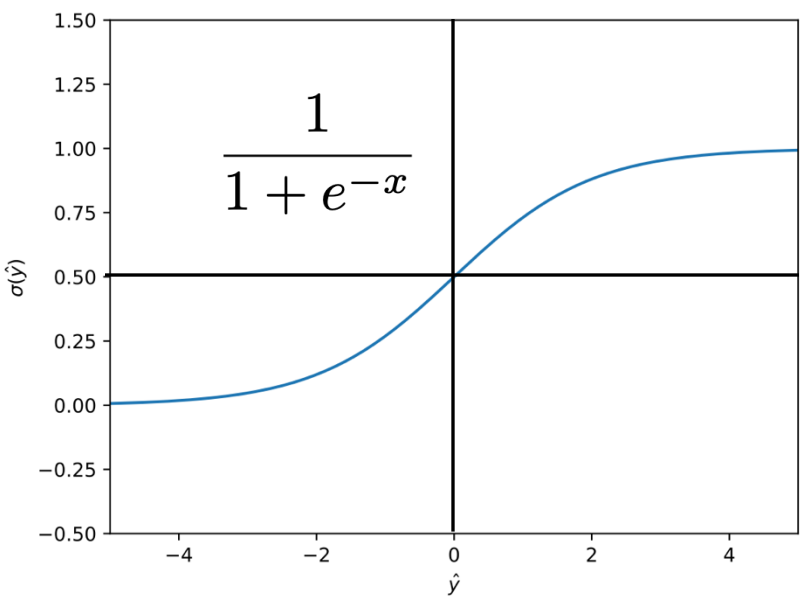
$$z^{(i)} = \mathbf{x}^{(i)\top}\mathbf{w}$$

- But logits $z \in \mathbb{R}$ must be “squashed" between 0 and 1
  - Sigmoid function is used

$$\sigma : \mathbb{R} \longrightarrow [0, 1]$$

Ingredient #1

$$p^{(i)} = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$$



Logistic Regression

- Model computes “logits" similar to linear regression:

$$z^{(i)} = \mathbf{x}^{(i)\top}\mathbf{w}$$

- But logits $z \in \mathbb{R}$ must be “squashed" between 0 and 1
  - Sigmoid function is used

$$\sigma : \mathbb{R} \longrightarrow [0, 1]$$

Ingredient #1

$$p^{(i)} = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$$

- Final prediction computed by thresholding (e.g., at 0.5):

$$\hat{y}^{(i)} = \begin{cases} 0 & \text{if } p^{(i)} < 0.5, \\ 1 & \text{if } p^{(i)} \geq 0.5 \end{cases}$$

Logistic Regression: loss function

- Denote dataset by $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$, where $y^{(i)}$ is 0 or 1.
- Logistic regression computes probabilities of class 1 as:

$$p^{(i)} = (1 + \exp(-z^{(i)}))^{-1} = (1 + \exp(-\mathbf{x}^{(i)\top}\mathbf{w}))^{-1}$$

- MSE does not work well with Logistic Regression. Instead, **binary cross-entropy** (aka log-loss) is used for loss function:

Ingredient #2

$$L(\mathbf{y}; \mathbf{x}; \mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)} \log p^{(i)} + (1 - y^{(i)}) \log(1 - p^{(i)})]$$

Why not to use MSE as loss function?

Details

Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
  - σ(*z*) + σ(-*z*) = 1

$$
\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
1-\sigma(z) &= 1-\frac{1}{1+e^{-z}} \\
&= \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} \\
&= \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \\
&= \frac{1}{1+e^{z}} \\
&= \sigma(-z)
\end{aligned}
$$

Details

Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
  - σ'(*z*) = σ(*z*)(1 - σ(*z*))

$$
\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
\frac{\partial \sigma}{\partial z} = \sigma'(z) &= -\frac{1}{(1+e^{-z})^2}(e^{-z} \times (-1)) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{e^{-z}}{1+e^{-z}} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1+e^{z}} \times \frac{1}{1+e^{-z}} \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}
$$

Logistic regression

How to train?

- **In the next slides, we will see that MSE is a bad loss function.**
- Unlike linear regression, logistic regression+MSE has no analytical (closed-form) solution.
  - We can use (stochastic) gradient descent instead.
  - We have to apply the **chain-rule of differentiation** to handle the sigmoid function.

Gradient descent for logistic regression

Details

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&=
\end{aligned}
$$

Gradient descent for logistic regression

Details

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)\sigma(\mathbf{x}^\top \mathbf{w})\left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right) \\
&= \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &\overset{14}{=} \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)\sigma(\mathbf{x}^\top \mathbf{w})\left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right) \\
&= \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})
\end{aligned}
$$

Notice the extra multiplicative terms compared to the gradient for *linear* regression: x($\hat{y}$ - y)

Attenuated gradient

- What if the weights **w** are initially chosen badly, so that $\hat{y}$ is very close to 1, even though $y = 0$ (or vice-versa)?
  - Then $\hat{y}(1 - \hat{y})$ is close to 0.
- In this case, the gradient:

$$\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{w}) = \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})$$

  will be very close to 0.
- If the gradient is 0, then no learning will occur!


TRAINING CLASSIFIERS USING MSE
imgflip.com

Working out a different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.
- One particularly well-suited cost function uses **logarithms**.
- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}} \left[\log \sigma(\mathbf{x}^\top \mathbf{w}) \right] =$$

Working out a different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.
- One particularly well-suited cost function uses **logarithms**.
- Logarithms and the logistic sigmoid interact well:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}}\left[\log \sigma(\mathbf{x}^\top \mathbf{w})\right] &= \mathbf{x}\frac{1}{\sigma(\mathbf{x}^\top \mathbf{w})}\sigma(\mathbf{x}^\top \mathbf{w})\left(1-\sigma(\mathbf{x}^\top \mathbf{w})\right) \\ &= \mathbf{x}\left(1-\sigma(\mathbf{x}^\top \mathbf{w})\right)\end{aligned}$$

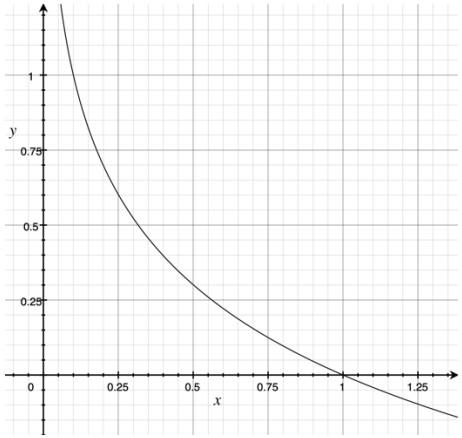
The gradient of log(σ) is surprisingly simple.

Log loss

- We want to assign a large loss when y=1 but $\hat{y}$=0

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y}$$



-log($\hat{y}$)

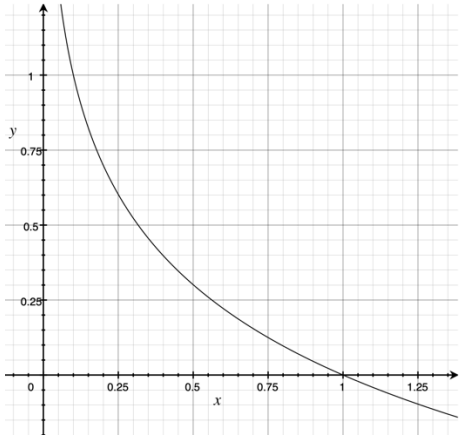
Log loss

- We want to assign a large loss when y=1 but $\hat{y}$=0, and for y=0 but $\hat{y}$=1.

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y} - (1-y) \log(1-\hat{y})$$

The y or (1-y) "selects" which term in the expression is active, based on the ground-truth label.



-log($\hat{y}$)

Details

Gradient descent for logistic regression with log-loss

$$\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) \quad = \quad \nabla_{\mathbf{w}} \left[- \left(y \log \hat{y} + (1-y) \log(1-\hat{y}) \right) \right]$$

Details

Gradient descent for logistic regression with log-loss

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} + (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1-\sigma(\mathbf{x}^\top \mathbf{w})) \right) \\
&= -\left(y \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{\sigma(\mathbf{x}^\top \mathbf{w})} - (1-y) \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{1-\sigma(\mathbf{x}^\top \mathbf{w})} \right) \\
&= -\left(y\mathbf{x}(1-\sigma(\mathbf{x}^\top \mathbf{w})) - (1-y)\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - y\sigma(\mathbf{x}^\top \mathbf{w}) - \sigma(\mathbf{x}^\top \mathbf{w}) + y\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - \sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= \mathbf{x}(\hat{y} - y)
\end{aligned}
$$

Same as for linear regression!

Recap

- Logistic Regression: used in binary classification tasks
  - Model

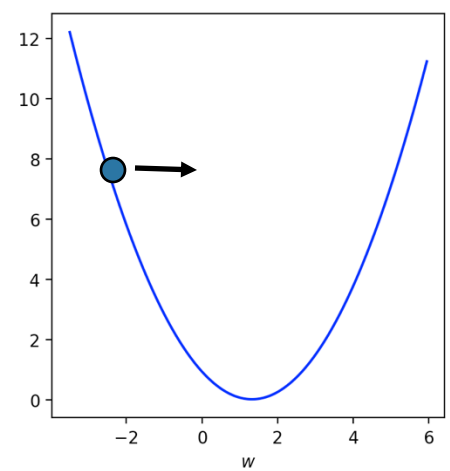
$$p^{(i)} = \sigma(z^{(i)}) = \frac{1}{1 + e^{-z^{(i)}}}$$

where $z^{(i)} = \mathbf{x}^{(i)\top} \mathbf{w}^{(i)}$

  - Loss: log-loss or binary cross-entropy loss (BCE)

$$L(\mathbf{y}; \mathbf{x}; \mathbf{w}) = -\frac{1}{n} \sum_{i=1}^{n} [y^{(i)} \log p^{(i)} + (1 - y^{(i)}) \log(1 - p^{(i)})]$$

- Gradient-based optimization
  - Gradient descent:
    w = w - ε ∂L/∂w
    where ε is learning rate.



Linear regression versus logistic regression

	Linear regression	Logistic regression
Primary use	Regression	Classification
Prediction ($\hat{y}$)	$\hat{y} = \mathbf{x}^T\mathbf{w}$	$\hat{y} = \sigma(\mathbf{x}^T\mathbf{w})$
Cost/Loss	MSE	Log-loss
Gradient	$\mathbf{x}(\hat{y} - y)$	$\mathbf{x}(\hat{y} - y)$

- Logistic regression is used primarily for *classification* even though it's called "regression".
- Logistic regression is an instance of a **generalized linear model** — a linear model combined with a **link function** (e.g., logistic sigmoid).
  - In DL, link functions are typically called **activation functions**.

Logistic Regression Trivia

Mark ALL true sentences about Logistic Regression:

- is used for regression tasks
- is trained with the log-loss function
- has unique local optimum (yes; beyond scope)
- has a closed-form solution

Multi-class classification

- So far we have talked about *binary classification.*

- But there are many settings in which multiple (>2) classes exist, e.g., emotion recognition, hand-written digit recognition:

6 classes (fear, anger, sadness, happiness, disgust, surprise)

10 classes (0-9)

Based on slide from WPI CS 541 Jacob Whitehill

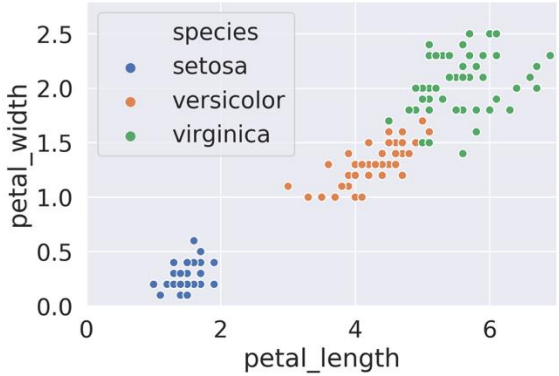
Multi-class classification

- It turns out that logistic regression can easily be extended to support an arbitrary number (≥2) of classes.
  - The multi-class case is called **softmax regression** or sometimes **multinomial logistic regression.**
- How to represent the ground-truth y and prediction $\hat{y}$?
  - Instead of just a scalar y, we will use a vector $\mathbf{y}$.

Based on slide from WPI CS 541 Jacob Whitehill

Multinomial Logistic Regression

- Denote dataset by $\mathcal{D} = \{(\mathbf{x}^{(i)}, \ell^{(i)})\}_{i=1}^{n}$, where $\ell^{(i)}$ is a label.
  - Example: measurements for Iris (flower) species



	sepal_length	sepal_width	petal_length	petal_width	species
	5.5	2.5	4.0	1.3	versicolor
	6.4	2.9	4.3	1.3	versicolor
	4.8	3.4	1.6	0.2	setosa
$(\mathbf{x}^{(i)}, \ell^{(i)})$	5.3	3.7	1.5	0.2	setosa
	6.7	2.5	5.8	1.8	virginica

- It is convenient to index the classes, to map $\ell^{(i)}$ to [1, …, K]. Then we can use one-hot encoding to obtain $y^{(i)}$.

Encoding multiple labels

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 2, 1.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{y}^{(3)} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

- Exactly 1 coordinate of each **y** is 1; the others are 0.

Based on slide from WPI CS 541 Jacob Whitehill

Encoding multiple labels

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 2, 1.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{y}^{(3)} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

This "slot" is for class 0.

- This is called a **one-hot encoding** of the class label.

Based on slide from WPI CS 541 Jacob Whitehill

Encoding multiple labels

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad \mathbf{y}^{(3)} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

This "slot" is for class 0.

This "slot" is for class 1.

This is called one-hot encoding

- Exactly 1 coordinate of each **y** is 1; the others are 0.

Based on slide from WPI CS 541 Jacob Whitehill

Prediction output

- The machine's predictions $\hat{\mathbf{y}}$ about each example's label are also **probabilistic**.

- They could consist of:

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.89 \\ 0.02 \\ 0.09 \end{bmatrix}$$

← **Machine's "belief" that the label is 0.**

← **Machine's "belief" that the label is 1.**

$$\hat{\mathbf{y}}^{(2)} = \begin{bmatrix} 0.43 \\ 0.51 \\ 0.06 \end{bmatrix}$$

- Each coordinate of $\hat{\boldsymbol{y}}$ is a probability. Sum of the coordinates in each $\hat{\mathbf{y}}$ is 1.

Based on slide from WPI CS 541 Jacob Whitehill

Softmax activation function

- Logistic regression outputs a *scalar* label $\hat{y}$ representing the probability that the label is 1.
  - We needed just a single weight vector $\mathbf{w}$, so that $\hat{y} = \sigma(\mathbf{x}^T\mathbf{w})$.

- Softmax regression outputs a *K-vector* representing the probabilities that the label is k=1, …, K.
  - We need K different vectors of weights $\mathbf{w}_1$, …, $\mathbf{w}_K$.
  - Weight vector $\mathbf{w}_k$ computes how much input $\mathbf{x}$ "agrees" with class k.

Based on slide from WPI CS 541 Jacob Whitehill

Softmax activation function

- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}_1$$

$$z_2 = \mathbf{x}^\top \mathbf{w}_2$$

$$\vdots$$

$$z_K = \mathbf{x}^\top \mathbf{w}_K$$

Recall: we refer to the z's as "logits".

- We then **normalize** across all *K* classes so that:
  1. Each output $\hat{y}_k$ is non-negative.
  2. The sum of $\hat{y}_k$ over all *K* classes is 1.

Based on slide from WPI CS 541 Jacob Whitehill

Multinomial Logistic Regression

- Logits for each class *k* are

$$z_k^{(i)} = \mathbf{x}^{(i)\top} \mathbf{w}_k$$

- Probabilities for class *k* computed using softmax function:

Ingredient #1

$$p_k^{(i)} = \frac{\exp(z_k^{(i)})}{\sum_{k'=1}^{K} \exp(z_{k'}^{(i)})}$$

$$\mathrm{Softmax}(\mathbf{z}) = \frac{e^{\mathbf{z}}}{\sum_k e^{z_k}}$$

- Loss function: cross-entropy (generalizes log-loss):

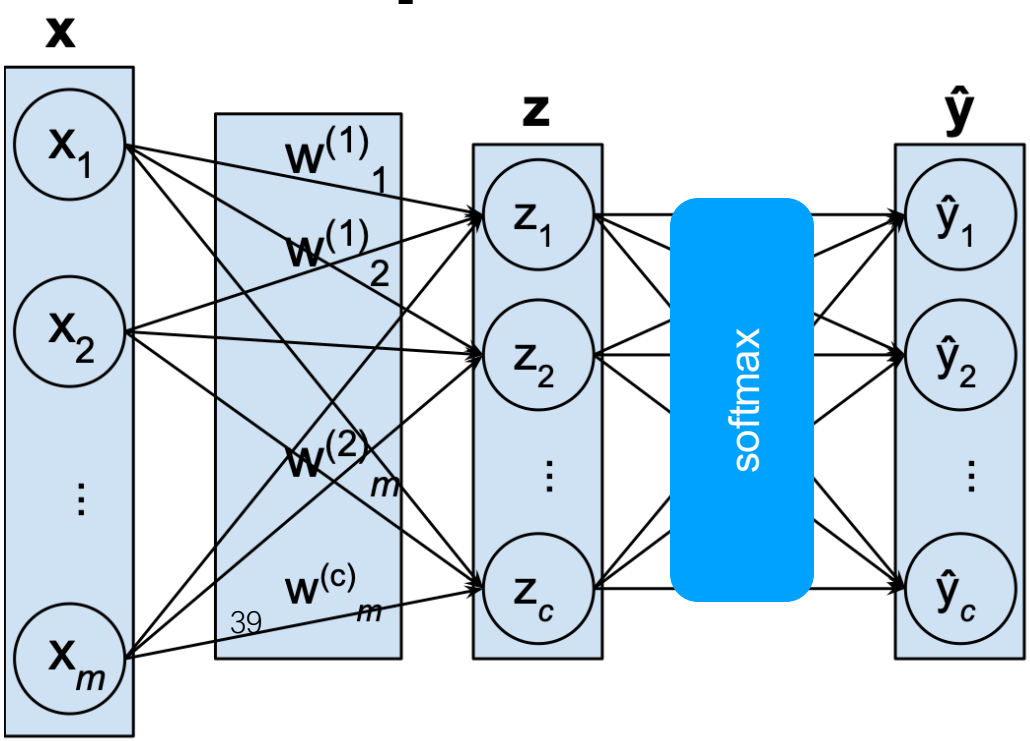
Ingredient #2

$$L(\mathbf{y}; \mathbf{x}; \mathbf{W}) = -\frac{1}{n} \sum_{k=1}^{K} \sum_{i=1}^{n} y_k^{(i)} \log p_k^{(i)}$$

- Optimized using gradient-based methods

Ingredient #3

Softmax regression: visual representation



Softmax regression

- In HW2, you will apply softmax regression to train a **handwriting recognition system** that can recognize all 10 digits (0-9).

- You will use the popular FashionMNIST dataset consisting of 60K training examples and 10K testing examples:

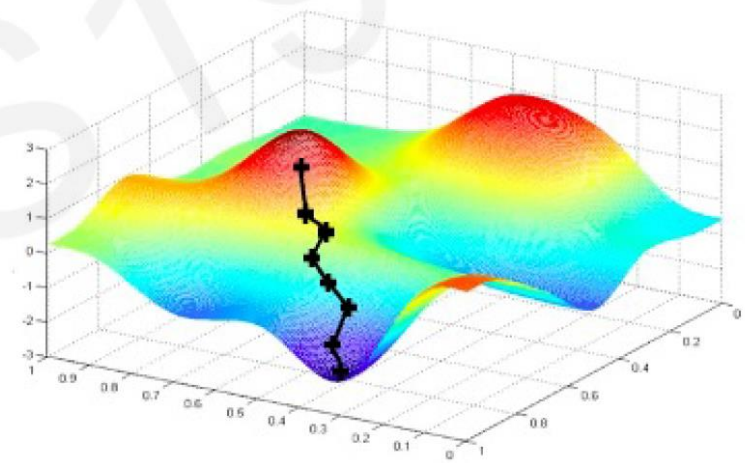
Stochastic Gradient Descent

Credits: first slides based on MIT 6.S191

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\boxed{\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

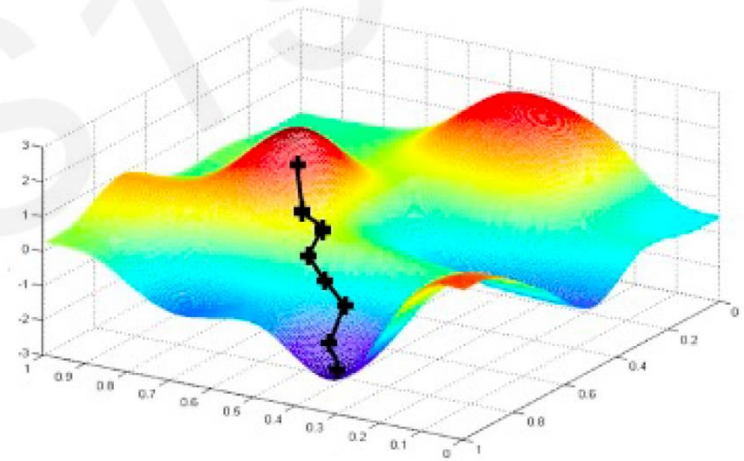


Computes loss gradients over **entire training data**; can be very **computationally intensive**!

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\boxed{\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

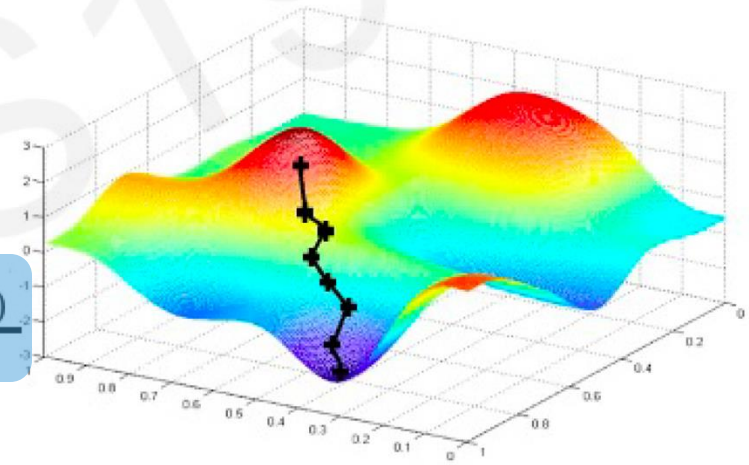


Easy to compute but
very noisy (stochastic)!

(Mini-batch) Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B}\frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of true gradient!

Mini-batches while training

- In comparison to pure SGD (i.e., B=1):
  - Mini-batch SGD **estimates gradient more accurately**
    - Smoother convergence
    - Allows for larger learning rates
  - Mini-batch SGD leads to faster training
    - Parallel computation achieves significant speedups on GPUs (Faster to compute gradient for mini-batch of size B than for B observations sequentially)
    - Model is likely to improve after each iteration (= processing each batch)

How to pick mini-batch size B?

1. Start from "small" initial value B
2. Train model for a couple of iterations
3. If program doesn't crash for running out of memory then DOUBLE the mini-batch size B
4. Repeat until program crashes
5. Go back to previous value of B (that worked)

"Small" is a value that doesn't crash your program; depends on model size and memory

Important: A good rule of thumb is to increase learning rate η proportionally to mini-batch size (so if you tuned η before tuning B, you will need to change η again for best results).

"Sampling" description of SGD

- This is the idea behind **stochastic gradient descent** (SGD):
  - Randomly <u>sample</u> a small ($\ll n$) **mini-batch** (or sometimes just **batch**) of training examples.
  - <u>Estimate</u> the gradient on just the mini-batch.
  - <u>Update</u> weights based on *mini-batch* gradient estimate.
  - Repeat.

Is expected value of stochastic gradient equal to gradient?

In practice, we don't "sample": shuffle and iterate over mini batches

"Shuffling" description of SGD

- In practice, SGD is usually conducted over multiple epochs.
  - An **epoch** is a single pass through the entire training set.
- Procedure:
  1. Let $\tilde{n} \ll n$ equal the size of the mini-batch.
  2. For e = 0 to numEpochs:
     I. Randomize the order of the examples in the training set.
     II. For i = 0 to $\tilde{n} \ll n$ (one epoch):
        A. Select a mini-batch $\mathcal{J}$ containing the next $\tilde{n}$ examples.
        B. Compute the gradient on this mini-batch: $\frac{1}{\tilde{n}} \sum_{i \in \mathcal{J}} \nabla_{\mathbf{w}} f(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}; \mathbf{W})$
        C. Update the weights based on the current mini-batch gradient.

What is the advantage of shuffling in every epoch?

SGD: variable learning rate

- One common learning rate "schedule" is to multiply ε by $c \in (0, 1)$ after every epoch.
  - This is called **exponential decay**.
- Another possibility (which avoids the issue) is to set the number of epochs T to a finite number.
  - SGD may not fully converge, but the machine might still perform well.
- There are many other strategies.

Pro tip: watch the loss function

Stochastic gradient descent

- Despite the “noise” (statistical inaccuracy) in the mini-batch gradient estimates, we often converge to good parameterizations.

- Reaching a “low" loss value can be much faster than regular gradient descent. Why?

  - Because we adjust the weights *many times* per epoch.

Advanced

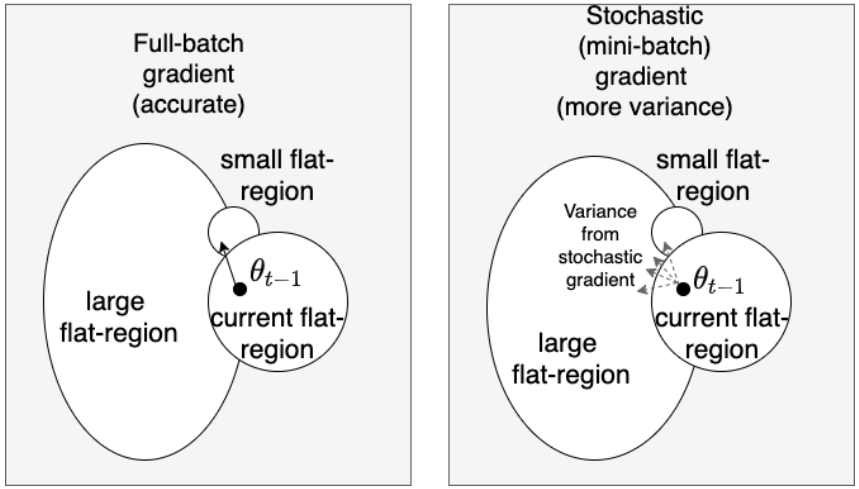
Why SGD matters?

- We often think of SGD as workaround for training with large data (cannot load it all in memory at once)
- In practice, it also leads to weights that **generalize better** at test time than full-batch gradient descent. Why?
  - We do not fully understand yet!
  - A lot of the explanations are speculative, but with mounting empirical evidence

Advanced

Hypothesis: SGD prefers large flat minima regions

- **Full-batch gradient** (entire dataset) at step *t*-1: low-noise gradient, points to the lowest minima in the vicinity of θ_{t-1}, often a small region
- **SGD** at step *t*-1: noisy gradient, points to various lower-loss regions in the vicinity of θ_{t-1}.



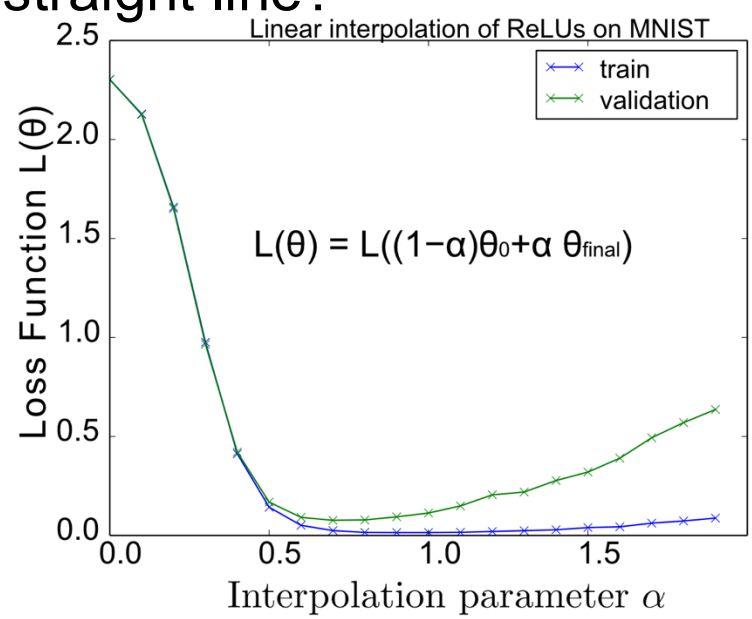
Credits: Bruno Ribeiro

Variance from SGD tends to move towards ***large flat regions****, preventing it from getting "stuck"*

Evidence of moving between flat regions

Advanced

- (Goodfellow et al., ICLR 2015) considers a MLP for handwritten digit classification.
- Let θ_0 be the initial (random) params and θ_f be the final parameters after SGD with very high accuracy.
- What the loss looks like if we interpolate θ_0 and θ_f using a straight line?



If flat regions were small, training loss should go up sometimes

Stochastic gradient descent

- ~~Despite~~ Thanks (!) to the “noise” (statistical inaccuracy) in the mini-batch gradient estimates, we often converge to good parameterizations.

- Reaching a certain loss value can be much faster than regular gradient descent because we adjust the weights *many times* per epoch.

Visual Examples of Training issues

1. Loss diverged
2. Loss too jittery
3. Stopped before convergence
4. Overfitting
5. Converges to different values every time

DONE ON THE WHITEBOARD

Optimization

- Like in the *Deep Learning* book, we define **optimization** as the algorithmic tools to help neural network training to reach a *lower loss value* during training.

Momentum

- SGD can suffer due to:
  - Noisy gradient estimates cause the weights to move in the wrong direction.
  - Slow convergence due to ill-conditioned loss function.
- Momentum is a commonly used technique to lessen these problems.

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) + \alpha \mathbf{V}, \quad \alpha \in [0, 1)$$

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

Previous steps

$$\mathbf{V}^{\text{new}} = \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) + \alpha \mathbf{V}, \quad \alpha \in [0, 1)$$

- i.e., actual weight update is a combination of a “moving average” of previous steps plus the current gradient estimate. *α* expresses how much we trust the average.

Momentum

- In SGD, instead of updating the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

we update them as:

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

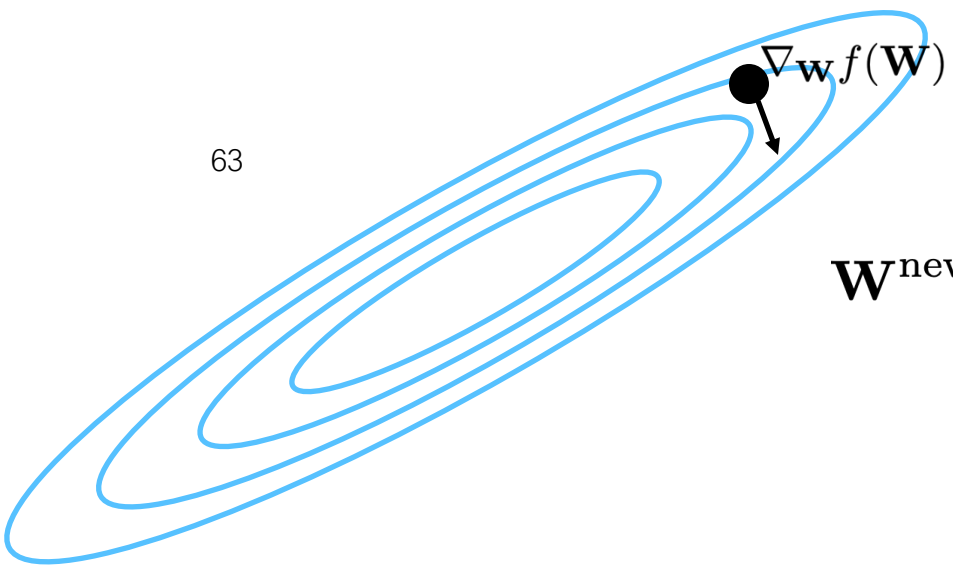
Current gradient estimate

$$\mathbf{V}^{\text{new}} = \epsilon \nabla_{\mathbf{W}} f(\mathbf{W}) + \alpha \mathbf{V}, \quad \alpha \in [0, 1)$$

- i.e., actual weight update is a combination of a “moving average” of previous steps plus the current gradient estimate. *α* expresses how much we trust the average.

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

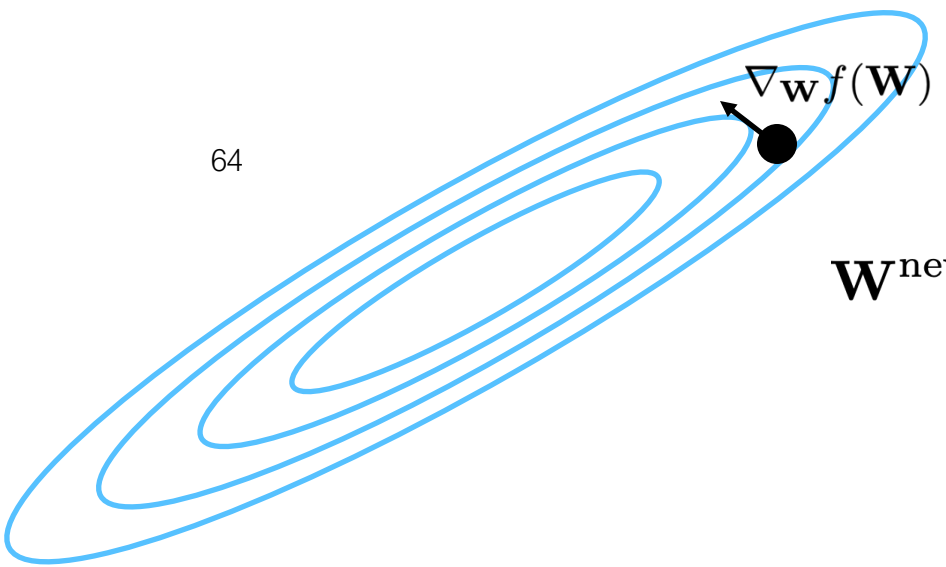


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

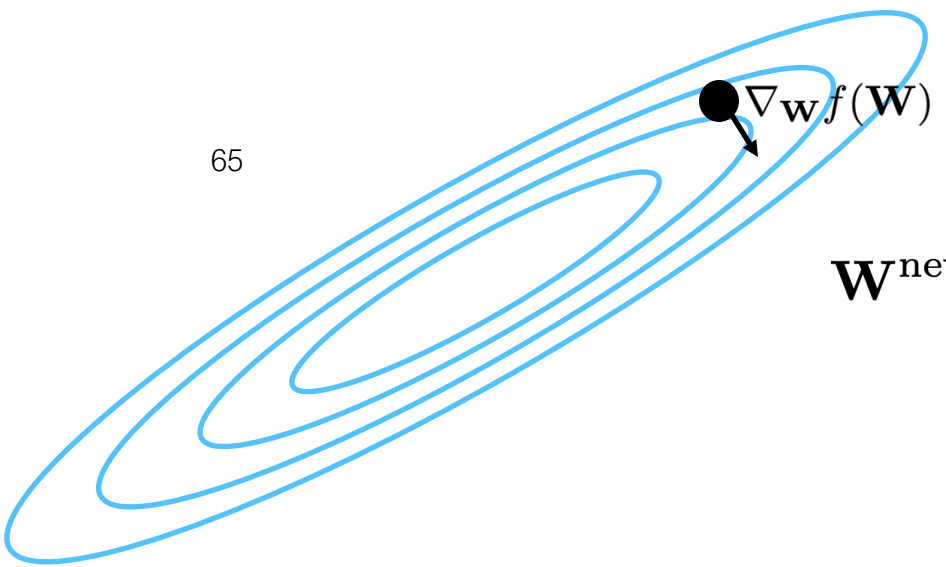


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

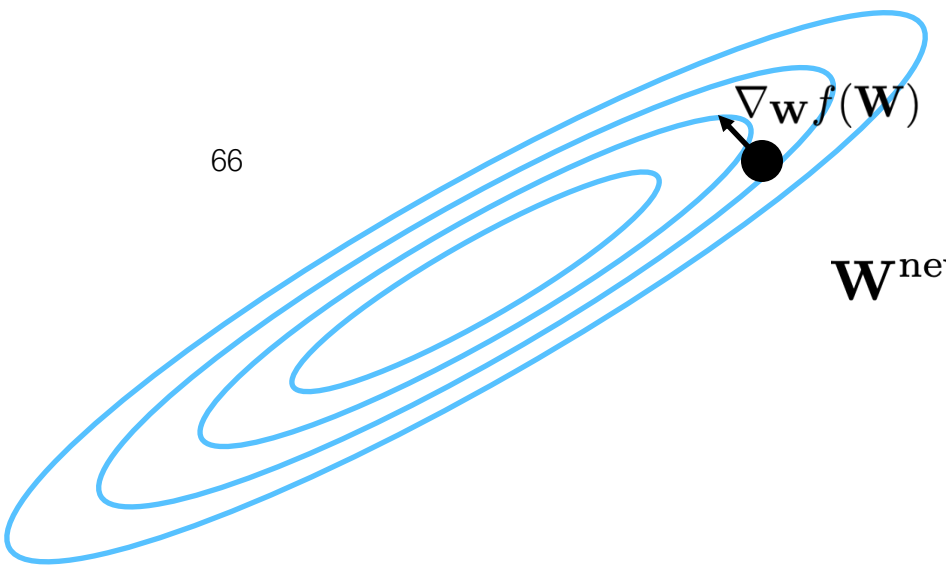


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.



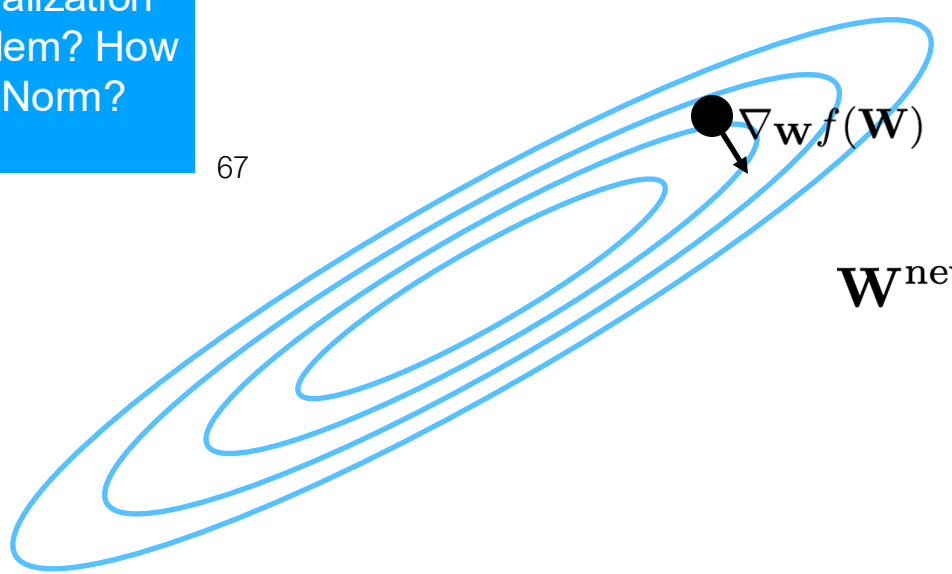
Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- Here we illustrate slow convergence with normal SGD due to an ill-conditioned loss function.

Could z-normalization solve this problem? How about BatchNorm?

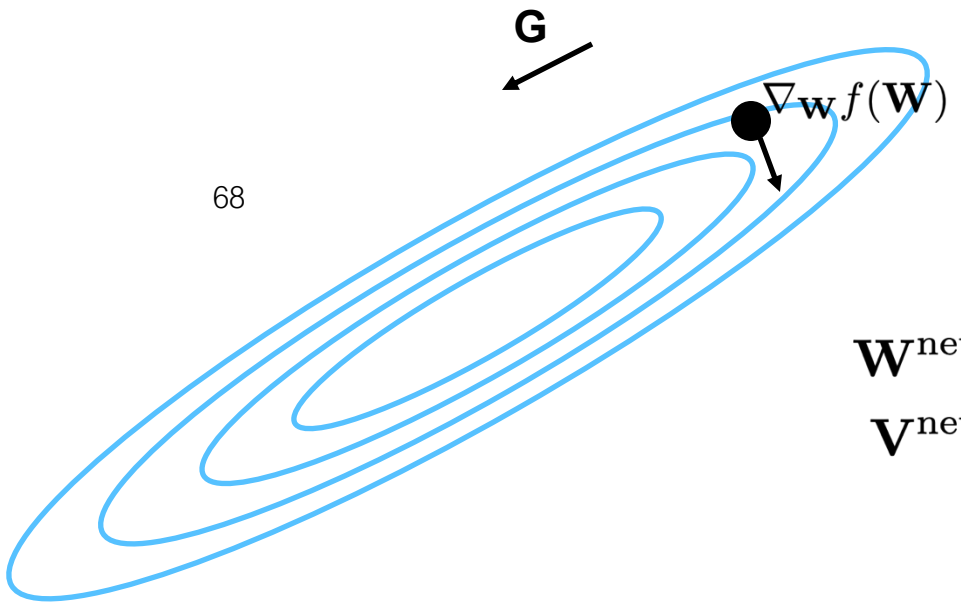


Without momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



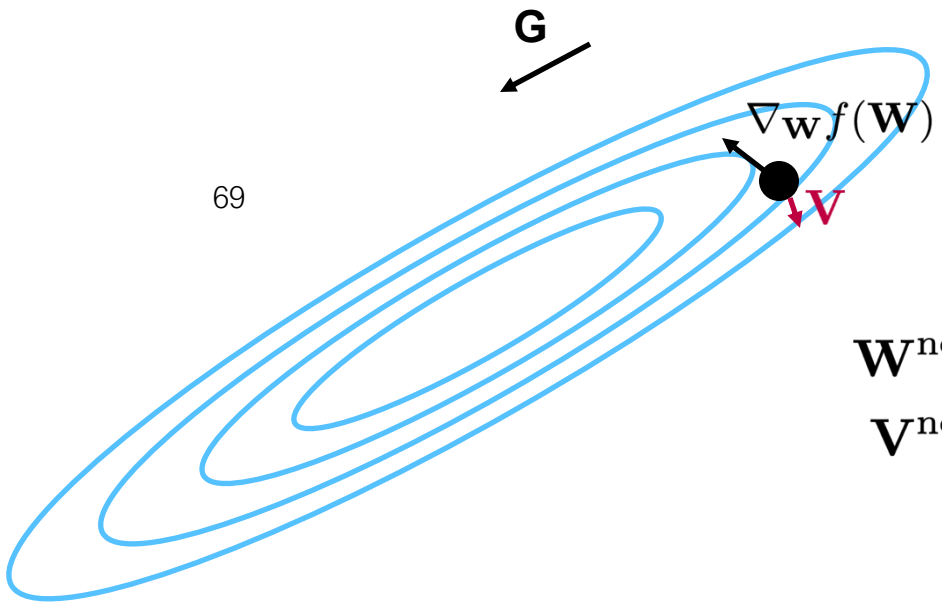
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



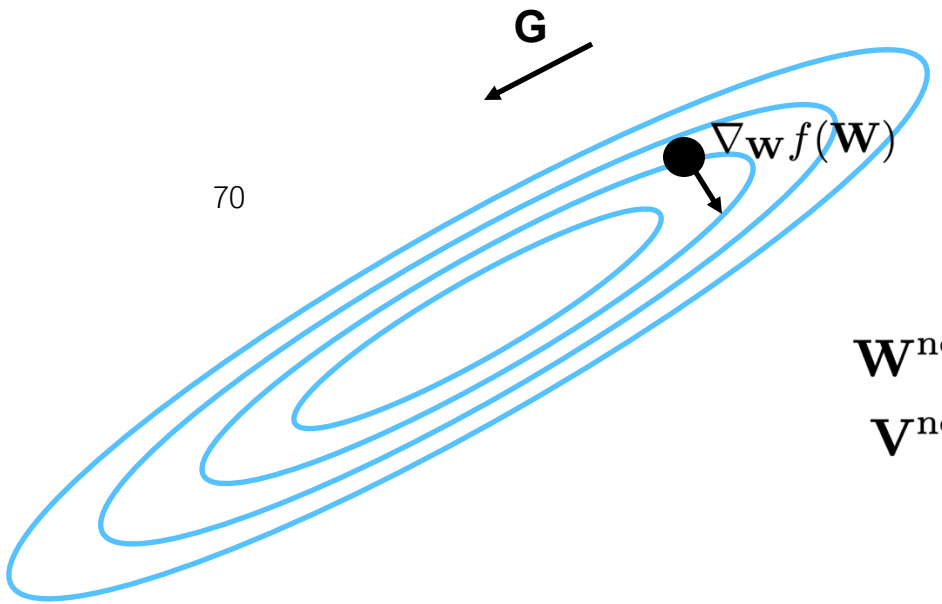
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



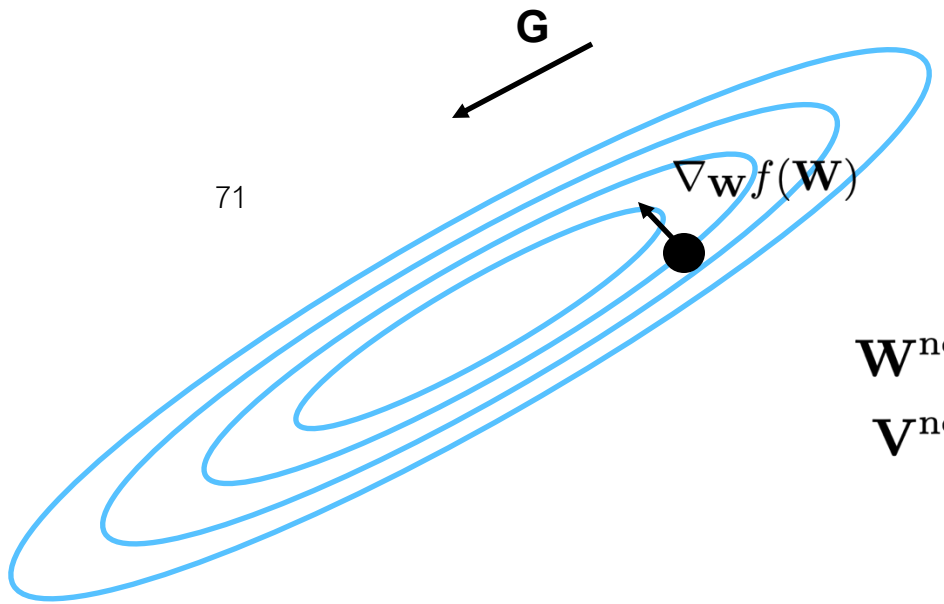
With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum

- If the $\nabla_{\mathbf{W}} f(\mathbf{W})$ consistently have positive projection along direction **G**, then the weights accumulate "momentum".



With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha \mathbf{V} + \epsilon \nabla_{\mathbf{W}} f(\mathbf{W})$$

Momentum in pytorch

torch.optim.SGD(params, lr=0.001, momentum=0, dampening=0, weight_decay=0, nesterov=False, *, maximize=False, foreach=None, differentiable=False, fused=None)

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

for $t = 1$ **to** $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  if $\mu \neq 0$
    if $t > 1$
      $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
    else
      $\mathbf{b}_t \leftarrow g_t$
    if *nesterov*
      $g_t \leftarrow g_t + \mu\mathbf{b}_t$
    else
      $g_t \leftarrow \mathbf{b}_t$
  if *maximize*
    $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

return θ_t

Can you recognize anything else?

What variable represents α?

With momentum

$$\mathbf{W}^{\text{new}} = \mathbf{W} - \mathbf{V}^{\text{new}}$$

$$\mathbf{V}^{\text{new}} = \alpha\mathbf{V} + \epsilon\nabla_{\mathbf{W}} f(\mathbf{W})$$