IMD1122
Special Topics in AI
Deep Learning

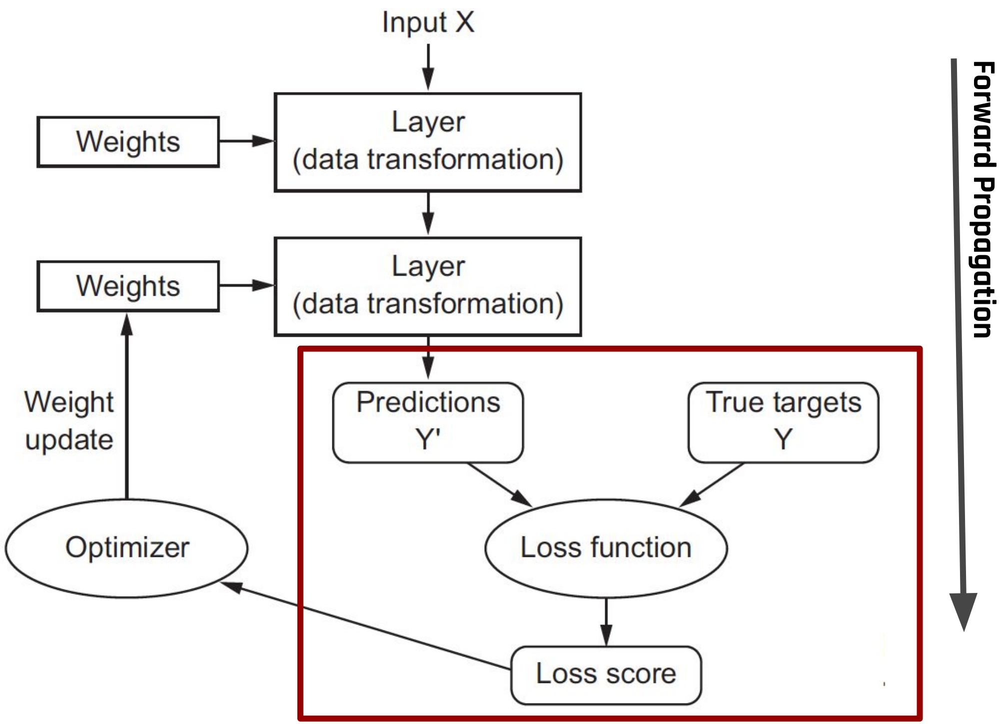
Training Neural Networks

Lesson #03

Training Neural Networks

Part #01

Understanding how DL works



Adapted from @FrançoisChollet

Loss Optimization

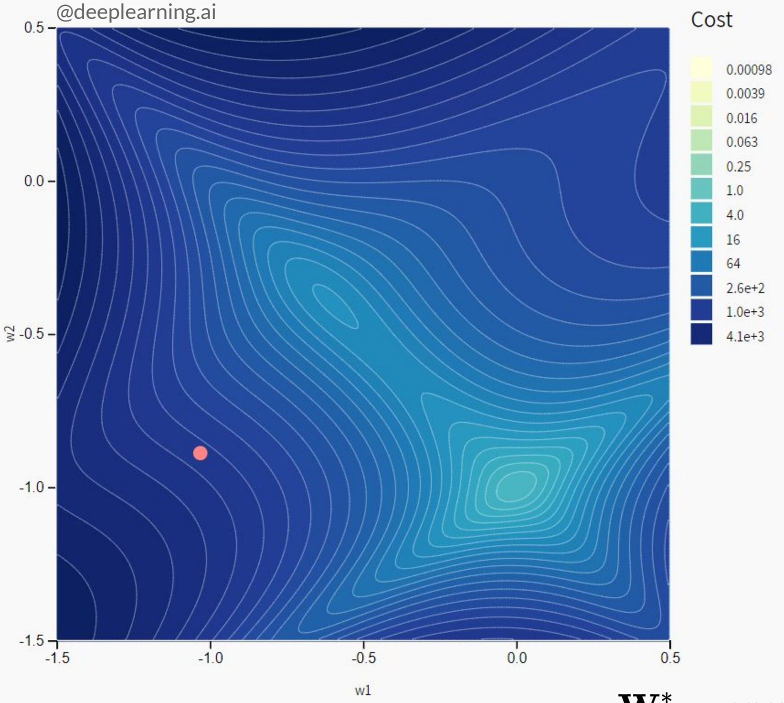
We want to find the network weights that **achieve the lowest loss**

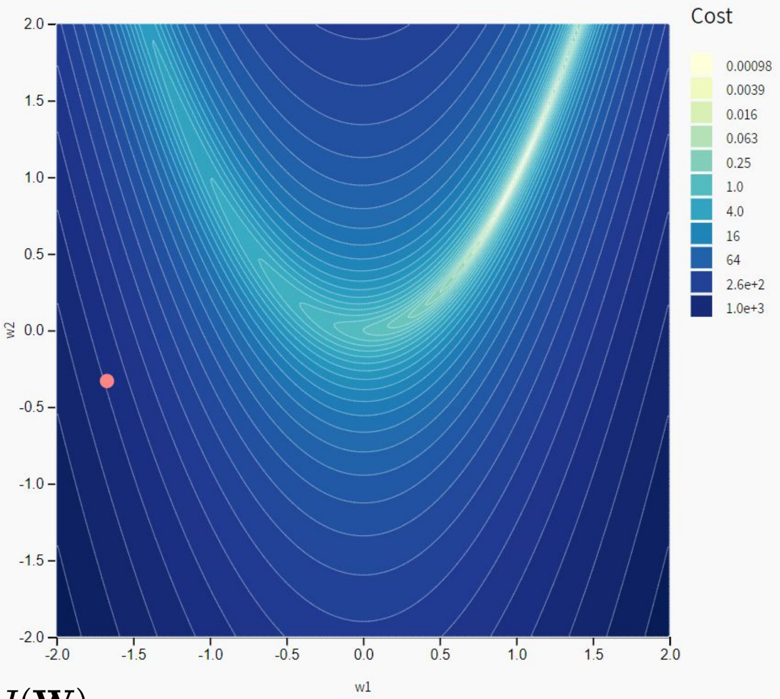
$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin} \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(f(X^{(i)}; \mathbf{W}), y^{(i)}\right)$$

$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin} \; J(\mathbf{W})$$

Loss Optimization

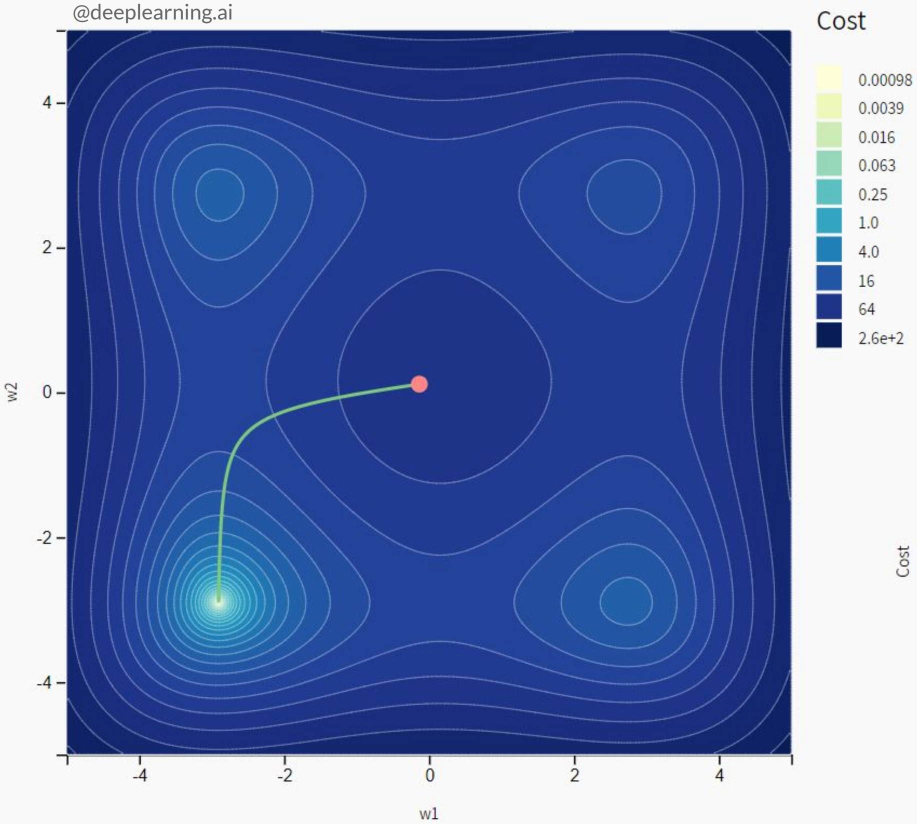
Remember: our loss is a function of the network weights!!!





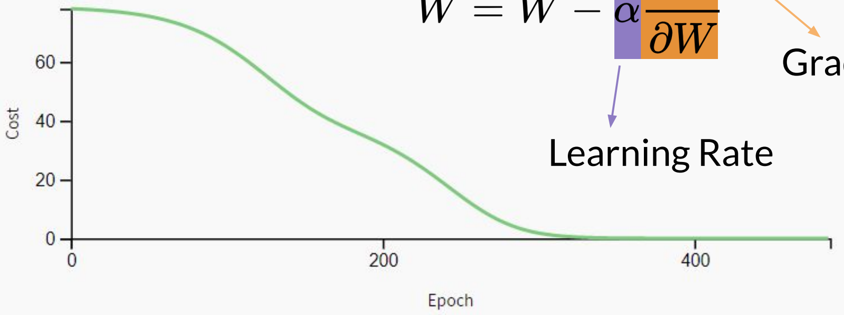
$$\mathbf{W}^* = \underset{\mathbf{W}}{argmin}\ J(\mathbf{W})$$

Loss Optimization - Gradient Descent

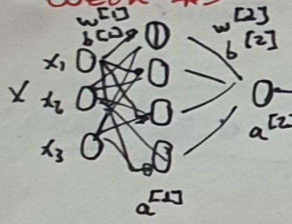


1. Initialize weights randomly
2. Compute gradient
3. Take small step in opposite direction of gradient
4. Repeat until convergence

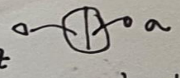
$$W = W - \underbrace{\alpha}_{\text{Learning Rate}} \underbrace{\frac{\partial J}{\partial W}}_{\text{Gradient}}$$



WEEK #3 Shallow Neural Networks



$y = a^{[2]}$



$n^{[0]}$ # inputs (Eg 3) $n^{[1]}$ # neurons (Eg 4) $n^{[2]}$ output (Eg 1)

X, $w^{[1]}$, $b^{[1]}$ → $\boxed{z^{[1]} = w^{[1]}x + b^{[1]}}$ → $\boxed{a^{[1]} = \sigma[z^{[1]}]}$ → $\boxed{z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}}$ ($w^{[2]}$, $b^{[2]}$) → $\boxed{a^{[2]} = \sigma[z^{[2]}]}$ → $\mathcal{L}(a^{[2]}, y)$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad (n_x, m)$$

features, from instances

$w^{[1]} = (n^{[1]}, n^{[0]})$

$b^{[1]} = (n^{[1]}, 1)$

$z^{[1]} = (n^{[1]}, 1)$

$x = (n^{[0]}, 1)$

$a^{[1]} = (n^{[1]}, 1)$

$w^{[2]} = (n^{[2]}, n^{[1]})$

$b^{[2]} = (n^{[2]}, 1)$

$z^{[2]} = (n^{[2]}, 1)$

$a^{[2]} = (n^{[2]}, 1)$

$$\mathcal{L}(a^{[2]}, y) = -y \log(a^{[2]}) - (1-y)\log(1-a^{[2]})$$

$$da^{[2]} = \frac{d\mathcal{L}}{da} = -\frac{y}{a^{[2]}} + \frac{(1-y)}{1-a^{[2]}} \quad (n^{[2]}, 1)$$

$$dz^{[2]} = \frac{d\mathcal{L}}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}} = \left[\frac{-y}{a^{[2]}} + \frac{(1-y)}{1-a^{[2]}}\right] a^{[2]}\cdot(1-a^{[2]})$$

$$\boxed{dz^{[2]} = a^{[2]} - y} \quad (n^{[2]}, 1)$$

$$\underbrace{dw^{[2]}}_{(n^{[2]}, n^{[1]})} = \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{dw^{[2]}} = \boxed{\begin{matrix} dz^{[2]} \cdot a^{[1]T} \\ (n^{[2]}, 1) \cdot (n^{[1]}, 1) \end{matrix}} \rightarrow dw^{[2]}$$

$$\underbrace{db^{[2]}}_{(n^{[2]}, 1)} = \frac{d\mathcal{L}}{dz^{[2]}} \cdot \frac{dz^{[2]}}{db^{[2]}} = \boxed{\begin{matrix} dz^{[2]} \\ (n^{[2]}, 1) \end{matrix}} \rightarrow db^{[2]}$$

$$\underset{(n^{[1]}, 1)}{da^{[1]}} = \frac{d\mathcal{L}}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} = \underset{(n^{[2]}, 1)}{dz^{[2]}} \cdot \underset{(n^{[2]}, n^{[1]})}{w^{[2]}}$$

$$\underbrace{w^{[2]T}}_{(n^{[1]}, n^{[2]})} \cdot \underbrace{dz^{[2]}}_{(n^{[2]}, 1)}$$

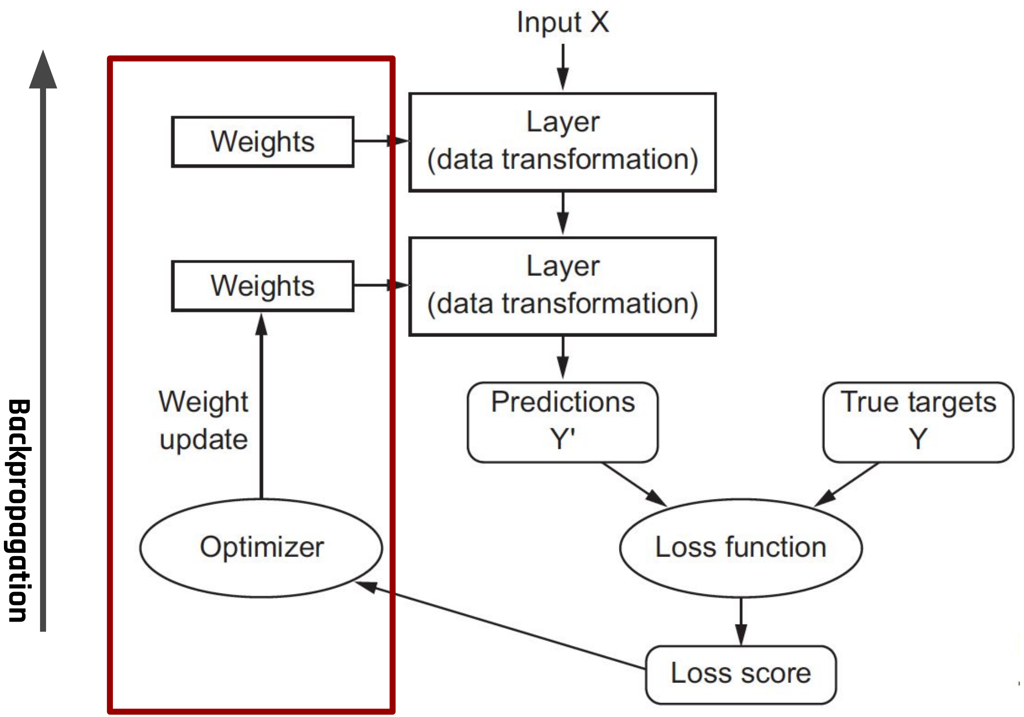
$$\underset{(n^{[1]}, 1)}{dz^{[1]}} = \frac{d\mathcal{L}}{da^{[1]}} \cdot \frac{da^{[1]}}{dz^{[1]}} = da^{[1]} \cdot g'(z^{[1]})$$

$$= (n^{[1]}, 1) * (n^{[1]}, 1) = \boxed{(w^{[2]T} \cdot dz^{[2]}) * g^{[1]\prime}(z^{[1]})} \rightarrow dz^{[1]}$$

$$\underset{(n^{[1]}, n^{[0]})}{dw^{[1]}} = \frac{d\mathcal{L}}{dz^{[1]}} \frac{dz^{[1]}}{dw^{[1]}} = \boxed{\underset{(n^{[1]}, 1)}{dz^{[1]}} \cdot \underset{(n^{[0]}, 1)}{x^T}} \rightarrow dw^{[1]}$$

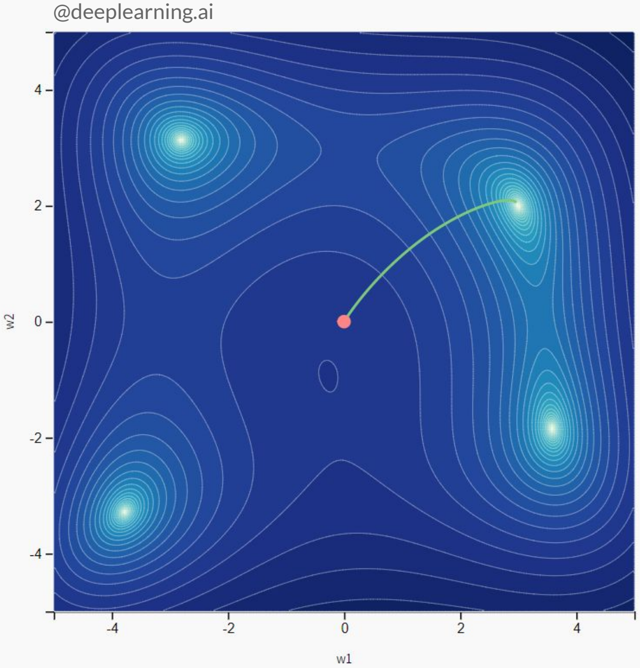
Understanding how DL works

Finding the right values of weights which minimize the error

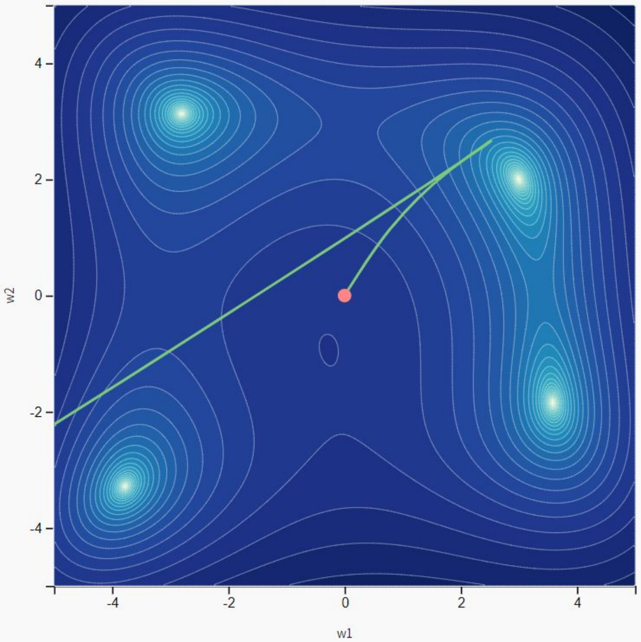


Adapted from @FrançoisChollet

Loss Functions Can Be Difficult to Optimize



Small learning rate (lr=0.001) converges slowly



Large learning rate (lr=0.1) overshoot, become unstable and diverge

How to deal with this?

Idea 1:

Try lots of different learning rates and see what works "just right"

Idea 2:

Do something smarter!!
Design a adaptive learning rate that "adapts" to the landscape

Optimization Algorithms

Algorithm

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

$$W = W - \alpha \boxed{?}$$

$\beta, \beta_1, \beta_2, learning\_decay$

TF Implementation

```
tf.keras.optimizers.SGD
```

```
tf.keras.optimizers.Adam
```

```
tf.keras.optimizers.Adadelta
```

```
tf.keras.optimizers.Adagrad
```

```
tf.keras.optimizers.RMSprop
```

Putting it all together

Mini-Batches

$1 \leq b \leq m$

```
# Create a source dataset from your training data
dataset = tf.data.Dataset.from_tensor_slices((train_set_x,train_set_y))
dataset = dataset.shuffle(buffer_size=64).batch(32)
```

Model

```
# Instantiate a simple classification model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(8, activation=tf.nn.relu, dtype='float64'),
  tf.keras.layers.Dense(8, activation=tf.nn.relu, dtype='float64'),
  tf.keras.layers.Dense(1, activation=tf.nn.sigmoid, dtype='float64')
])
```

Loss

Evaluation Metrics

```
# Instantiate a logistic loss function that expects integer targets.
loss = tf.keras.losses.BinaryCrossentropy()

# Instantiate an accuracy metric.
accuracy = tf.keras.metrics.BinaryAccuracy()
```

Optimizer

```
# Instantiate an optimizer.
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
```

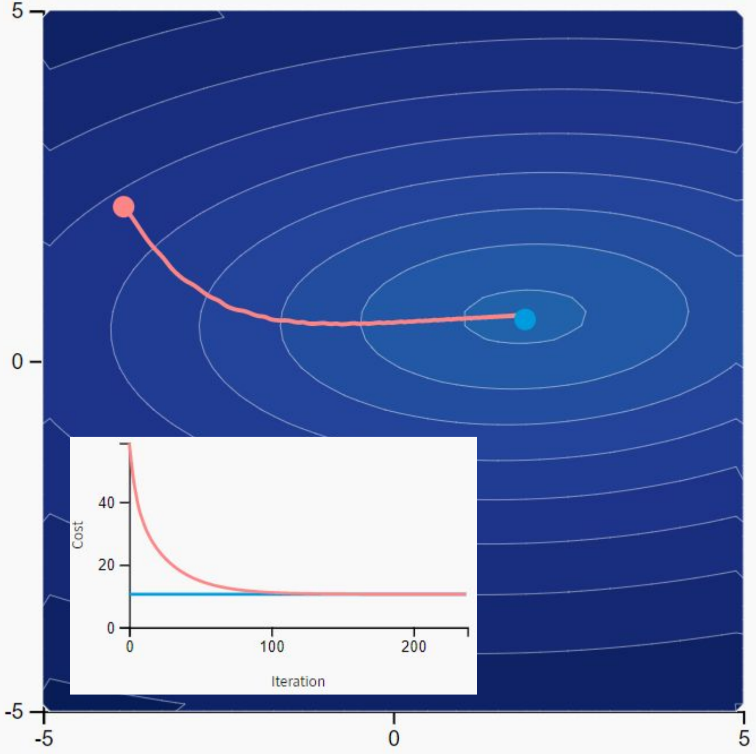
Putting it all together

```
for i in range(500):
    # Iterate over the batches of the dataset.
    for step, (x, y) in enumerate(dataset):
      # Open a GradientTape.
      with tf.GradientTape() as tape:

        # Forward pass.
        logits = model(x)

        # Loss value for this batch.
        loss_value = loss(y, logits)

      # Get gradients of loss wrt the weights.
      gradients = tape.gradient(loss_value, model.trainable_weights)

      # Update the weights of our linear layer.
      optimizer.apply_gradients(zip(gradients, model.trainable_weights))

      # Update the running accuracy.
      accuracy.update_state(y, logits)
```

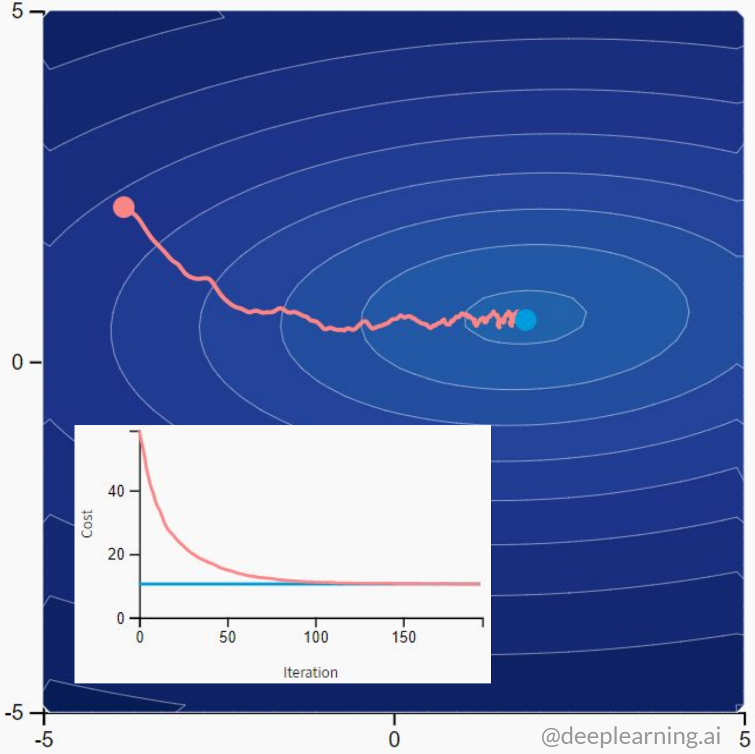
$$W = W - \alpha \frac{\partial J}{\partial W}$$

Mini-Batches Challenges

$m_b = m$



$m_b = 24$



Training Neural Networks

Part #02

TRAIN A DEEP NEURAL NETWORK

=

MINI-BATCH

+

OPTIMIZATION ALGORITHMS

→ Update W and b

Optimization Algorithms

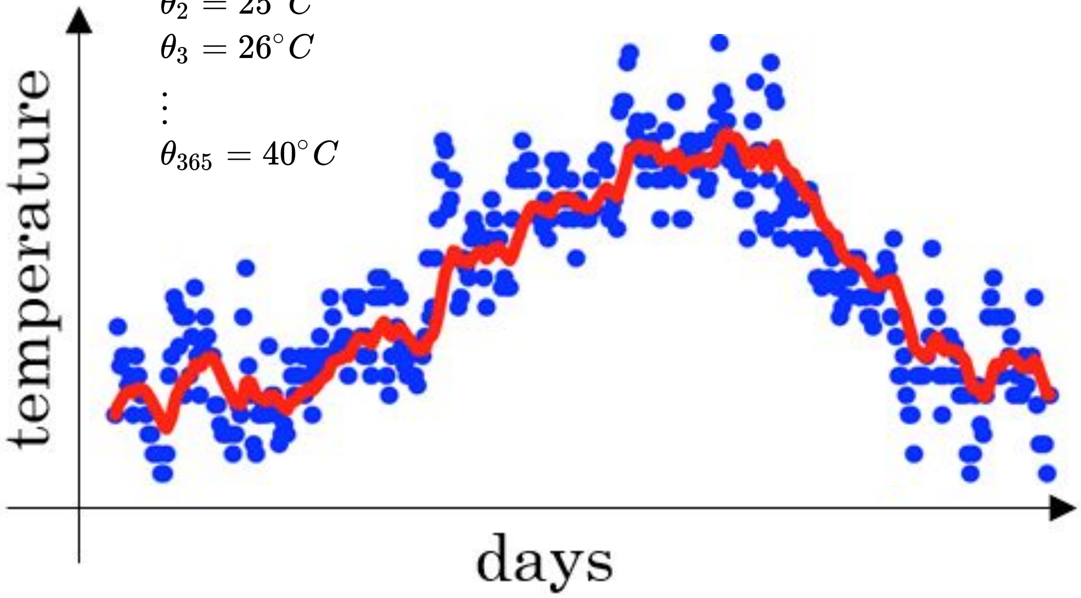
Exponentially Weighted Average

Adam

- Momentum
- RMSprop

Exponentially Weighted Average

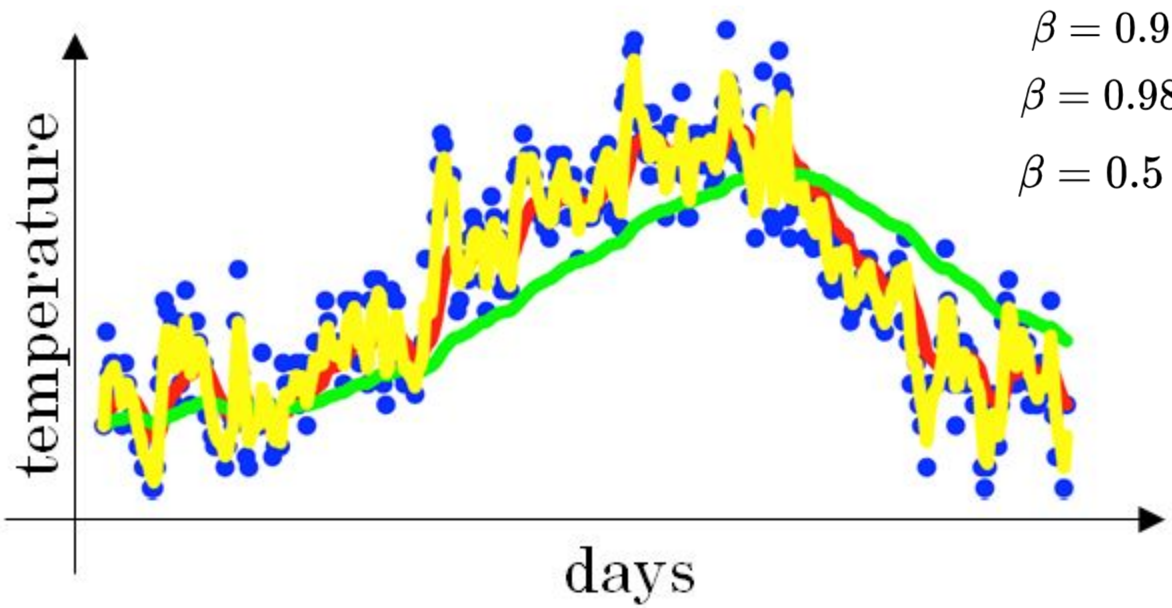
$$\theta_1 = 20^\circ C$$
$$\theta_2 = 25^\circ C$$
$$\theta_3 = 26^\circ C$$
$$\vdots$$
$$\theta_{365} = 40^\circ C$$



$$V_0 = 0$$
$$V_1 = 0.9V_0 + 0.1\theta_1$$
$$V_2 = 0.9V_1 + 0.1\theta_2$$
$$V_3 = 0.9V_2 + 0.1\theta_3$$
$$\vdots$$
$$V_{365} = 0.9V_{364} + 0.1\theta_{365}$$

$$V_t = 0.9V_{t-1} + 0.1\theta_t$$

Exponentially Weighted Average



$$\beta = 0.9 \rightarrow V_t \approx 10\ days\ of\ temperature$$

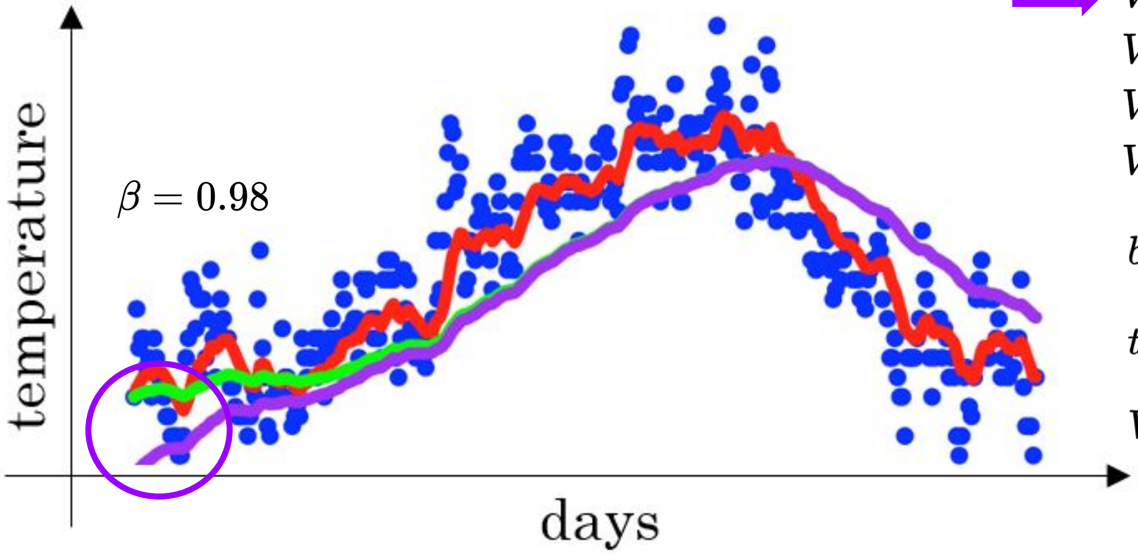
$$\beta = 0.98 \rightarrow V_t \approx 50\ days\ of\ temperature$$

$$\beta = 0.5 \rightarrow V_t \approx 2\ days\ of\ temperature$$

$$V_t = \beta V_{t-1} + (1-\beta)\theta_1$$

$$V_t \approx \frac{1}{1-\beta}\ days\ of\ temperature$$

Exponentially Weighted Average Bias Correction



$$V_0 = 0$$
$$V_1 = 0.98V_0 + 0.02\theta_1$$
$$V_2 = 0.98V_1 + 0.02\theta_2$$
$$V_2 = 0.98 * 0.02\theta_1 + 0.02\theta_2$$
$$V_2 = 0.0196\theta_1 + 0.02\theta_2$$

$$bias\_correction = \frac{V_t}{1 - \beta^t}$$

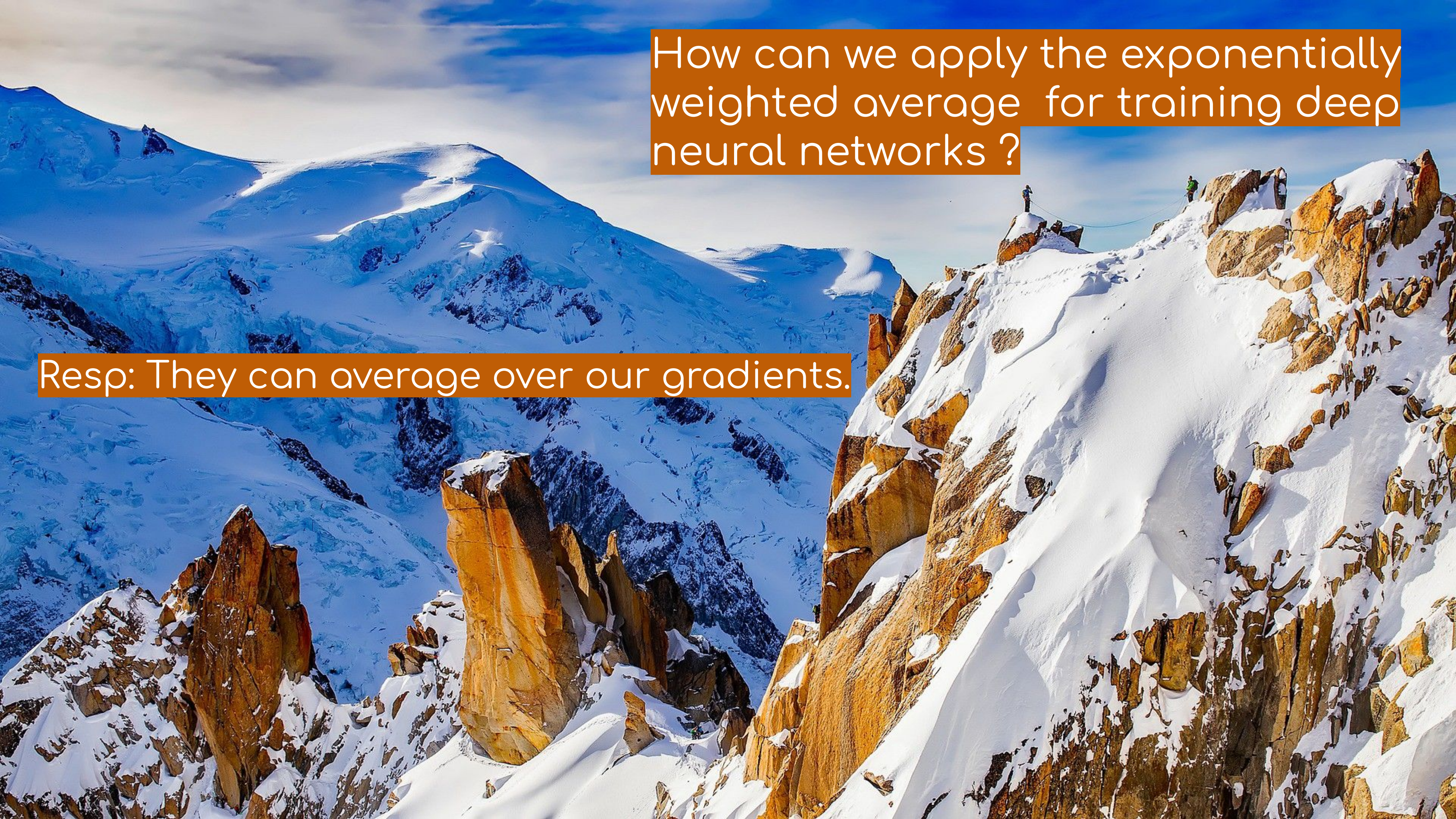
$$t = 2 \rightarrow 1 - \beta^2 = (1 - 0.98^2) = 0.0396$$

$$V_2 = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$

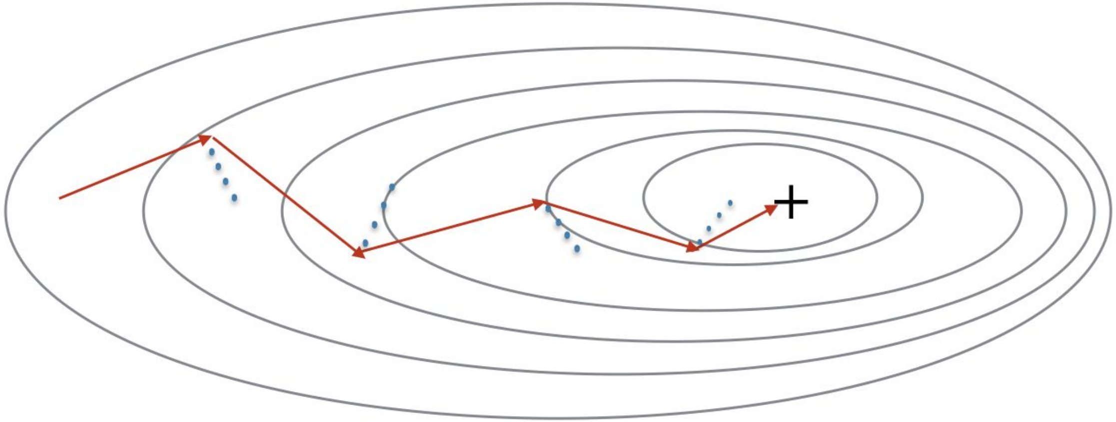
PETROBRAS PN N2 · D · BMFBOVESPA O29.51 H29.68 L29.03 C29.14 −0.78 (−2.61%)
EMA 50 close 29.37
SELL 29.08
0.09
BUY 29.17
Delayed
32.00
31.50
31.00
30.50
30.00
29.50
29.14
28.50
28.00
27.50
27.00
26.50
26.00
25.50
25.00
24.50
24.00
23.50
23.00
22.50
E
D
E
D
D
E
Jul
Aug
Sep
Oct
Nov
Dec
2020
Feb
Mar

How can we apply the exponentially weighted average for training deep neural networks ?

Resp: They can average over our gradients.



Gradient Descent with Momentum



- Momentum takes into account the past gradients to smooth out the update.
- Formally, this will be the exponentially weighted average of the gradient on previous steps.

Gradient Descent with Momentum

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta) dW$$

$$v_{db} = \beta v_{db} + (1-\beta) db$$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β $\qquad \beta = 0.9$

Gradient Descent with RMSprop

On iteration t:

Compute dW, db on the current mini-batch

$$s_{dW} = \beta s_{dW} + (1 - \beta) dW^2$$

$$s_{db} = \beta s_{db} + (1 - \beta) db^2$$

$$W = W - \alpha \frac{dW}{\sqrt{s_{dW}} + \varepsilon} \qquad b = b - \alpha \frac{db}{\sqrt{s_{db}} + \varepsilon}$$

$\varepsilon = 10$^(-8)

Hyperparameters: α, β $\qquad \beta = 0.9$

Gradient Descent with Adam

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW \qquad v_{db} = \beta v_{db} + (1-\beta)db$$

$$s_{dW} = \beta s_{dW} + (1-\beta)dW^2 \qquad s_{db} = \beta s_{db} + (1-\beta)db^2$$

$$v_{dW}^{correct} = v_{dW}/(1-\beta_1^t) \qquad v_{db}^{correct} = v_{db}/(1-\beta_1^t)$$

$$s_{dW}^{correct} = s_{dW}/(1-\beta_2^t) \qquad s_{db}^{correct} = s_{db}/(1-\beta_2^t)$$

$$W = W - \alpha \frac{v_{dW}^{correct}}{\sqrt{s_{dW}^{correct}} + \varepsilon} \qquad b = b - \alpha \frac{v_{db}^{correct}}{\sqrt{s_{db}^{correct}} + \varepsilon}$$

$\varepsilon = 10$^(-8)

Hyperparameters: α, β_1, β_2 $\quad \beta_1 = 0.9, \beta_1 = 0.999$

Learning Rate Decay

Idea: reduce α for each mini-bach t in order to smooth the gradient descent.

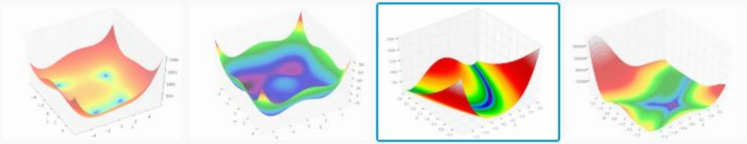
$$\alpha = \frac{1}{1 + learning\_decay \; * \; epoch\_num} * \alpha_0$$

$$\alpha = 0.95 \; e^{epoch\_num} \alpha_0$$

$$\alpha = \frac{k}{\sqrt{epoch\_num}} * \alpha_0$$

1. Choose a cost landscape

Select an artificial landscape $\mathcal{J}(w_1, w_2)$.



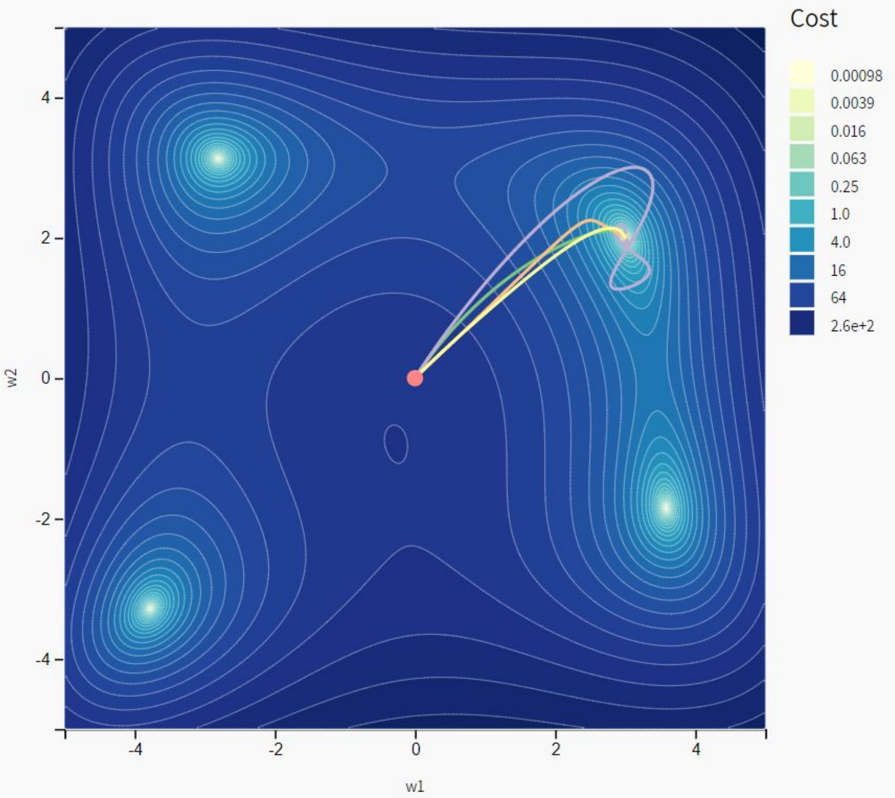
2. Choose initial parameters

On the cost landscape graph, drag the red dot to choose initial parameter values and thus the initial value of the cost.

3. Choose an optimizer

Select the optimizer(s) and hyperparameters.

Optimizer	Learning Rate	Learning Rate Decay
☑ Gradient Descent	0,001	0
☑ Momentum	0,001	0
☑ RMSprop	0,001	0
☑ Adam	0,001	0



https://www.deeplearning.ai/ai-notes/optimization/

Neural Networks in Practice #01

Splitting Data & Regularization

Next ….