Machine Learning Workshop 2

Variational Autoencoder

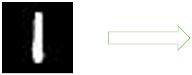
Jonathan Guymont

National Bank of Canada, 2018

Outline

Outline

An example to keep in mind



$$\begin{bmatrix} 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \mathbf{255} & \mathbf{211} & 111 & \cdots & 0 \\ 0 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 1 & \mathbf{246} & \mathbf{244} & 4 & \cdots & 0 \\ 0 & \cdots & 12 & \mathbf{233} & \mathbf{213} & 34 & \cdots & 0 \\ 0 & \cdots & 22 & \mathbf{252} & \mathbf{234} & 2 & \cdots & 0 \\ 0 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a0 & \cdots & 4 & \mathbf{233} & \mathbf{222} & 33 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Figure: Conversion of a greyscale image to an matrix

Note: In the workshop, we will mostly work with linear layer, so we also need to flatten the image.

```python
from PIL import Image
def image_to_array(image_path):
    with Image.open(image_path) as img:
        image = img.convert()
        array_image = np.asarray(image, np.float)
    return array_image
```

Outline

Autoencoders

Autoencoers are neural network that are trained to learn how to map their input to their input. Internally, it has an hidden layer $\boldsymbol{h}$ that contains a lossy summary of the relevant feature for the task.

Autoencoders

An autoencoder can be seen has a two parts network

- Encoder function: $\boldsymbol{z} = f_\phi(\boldsymbol{x})$
- Decoder function: $\tilde{\boldsymbol{x}} = g_\theta(\boldsymbol{z})$
- ϕ and θ are set of learned parameters

Autoencoders

The simplest autoencoder is a one layer MLP:

$$\begin{aligned} \mathbf{z} =& \mathrm{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \quad \text{[encoder]} \\ \tilde{\mathbf{x}} =& \mathrm{sigmoid}\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right) \quad \text{[decoder]} \end{aligned} \tag{1}$$

Pytorch simple autoencoder

```
class Autoencoder:
        def __init__(self, **kargs):
            """constructor"""
            pass

        def encoder(self, x):
            pass

        def decoder(self, z)
            pass

        def forward(self, x):
            pass
```

Parameter initialization

$$
\begin{aligned}
\mathbf{z} =& \text{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \\
\tilde{\mathbf{x}} =& \text{sigmoid}\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right)
\end{aligned}
\tag{2}
$$

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        # encoder parameters \phi
        self.Wxz = xavier_init(size=[x_dim, z_dim])
        self.bxz = Variable(torch.zeros(z_dim), requires_grad=True)
        # decoder parameters \theta
        self.Wzx = xavier_init(size=[z_dim, x_dim])
        self.bzx = Variable(torch.zeros(x_dim), requires_grad=True)
```

Encoder $f_\phi(x)$

$$\mathbf{z} = \text{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \tag{3}$$

$$\phi = \{\mathbf{W}_{xz}, \mathbf{b}_{xz}\} \tag{4}$$

```
class Autoencoder:
        ...
    def encoder(self, x):
        z = F.relu(self.Wxz @ z + self.bxz.repeat(x.size(0), 1))
        return z
```

Decoder $g_\theta(x)$

$$\mathbf{z} = \sigma\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right) \tag{5}$$

$$\boldsymbol{\theta} = \{\mathbf{W}_{zx}, \mathbf{b}_{zx}\} \tag{6}$$

```
class Autoencoder:
        ...
    def decoder(self, z):
        x_recon = F.sigmoid(z @ self.Wzx + self.bzx.repeat(z.size(0), 1))
        return x_recon
```

Forward propagation

$$\begin{aligned} \mathbf{z} =& \mathrm{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \\ \tilde{\mathbf{x}} =& \mathrm{sigmoid}\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right) \end{aligned} \tag{7}$$

```
class Autoencoder:
        ...
    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Pytorch simple autoencoder

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        # encoder parameters
        Wxz = xavier_init(size=[x_dim, z_dim])
        bxz = Variable(torch.zeros(z_dim), requires_grad=True)
        # decoder parameters
        Wzx = xavier_init(size=[h_dim, x_dim])
        bzx = Variable(torch.zeros(X_dim), requires_grad=True)

    def encoder(self, x):
        z = F.relu(x @ self.Wxh + self.bxh.repeat(x.size(0), 1))
        return z

    def decoder(self, z):
        x_recon = F.sigmoid(z @ self.Wzx + self.bzx.repeat(z.size(0), 1))
        return x_recon

    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Autoencoder - Loss Function

If you treat the problem like a *regression*[1], use the mean square error between the input and the reconstruction

$$\mathcal{L} = \sum_{i=1}^{d} (x_i - \tilde{x}_i)^2 \tag{8}$$

[1] If you do regression, you don't have to apply sigmoid in the decoder.

Training Autoencoders

Algorithm 1 Pseudocode for Stochastic Gradient Training

Require: Learning rate η
Require: Initial parameter $\boldsymbol{\omega}_0$
Require: Number of epochs T
  for $i = 1$ to T **do**
    $X = X^{train}$.copy() and $Y = Y^{train}$.copy()
    while X is not empty **do**
      Sample $\{x^{(1)}, ..., x^{(m)}\}$ from X and $\{y^{(1)}, ..., y^{(m)}\}$ from Y
      Remove samples from X and Y
      Compute gradient $\boldsymbol{g}_t = \frac{1}{m}\nabla_{\boldsymbol{\omega}} \sum_i \mathcal{L}(\tilde{x}^{(i)}, x^{(i)})$
      Apply update: $\omega_t = \boldsymbol{\omega}_{t-1} - \eta \cdot \boldsymbol{g}_t$
    end while
  end for

Pytorch Stochastic Gradient

```
def train(self, trainloader, num_epochs, learning_rate):

    for epoch in range(num_epochs):
        for inputs, targets in trainloader:
            batch_size = inputs.size(0)
            x_tilde = self.forward(x)

            loss = F.mse_loss(x, x_tilde)

            # Use autograd to compute the derivative of the loss w.r.t
            # all Tensors with requires_grad=True. After calling `loss.backward()`,
            # conv_weight.grad, dense_weight.grad, and dense_bias.grad
            # will be Tensors equal to the gradient of the loss with respect
            # to the filters of the cnn layer, the weight of the fully connected layer, and
            # the bias of the fully connected layer respectively.
            loss.backward()

            # Apply gradient descent to all the leaned parameters
            # The derivative of the loss is giving us the direction
            # where the funtion increase. Thus we go in the
            # opposite direction. Using torch.no_grad() tells pytorch
            # to not include thes operation in the computational graph.
            # Instead, gradient descent is goning to be applied `inplace`.
            with torch.no_grad():
                self.W_xz -= learning_rate * self.W_xz.grad
                self.b_xz -= learning_rate * self.b_xz.grad
                self.W_zx -= learning_rate * self.W_zx.grad
                self.b_zx -= learning_rate * self.b_zx.grad
```

Autoencoders

To summarize

- A neural network encode x in a hidden state z of smaller dimension
- Another neural network decode z to reconstruct x
- A sound loss function could be the mean square error between the input and its reconstruction.
- Both network can be train at the same time with gradient method.

Outline

Generative Models

Represent the probability distribution of either $P(X, Y)$ or $P(X)$. In the case of *density estimation*, we are looking for a representation of

$$x \sim P_\theta(X)$$

For example, $x \sim \mathcal{N}(x; \boldsymbol{\mu}_{\mathsf{mle}}, \boldsymbol{\sigma}_{\mathsf{mle}})$

Problem: Most parametric distribution make strong (and often wrong) assumption about the distribution.

Outline

Latent variable models

We can model the distribution of x as a function of a latent variable z

$$p(x) = \int p_\theta(x|z)p(z)dz$$

where the distribution of z is chosen. A typical choice for z is

$$z \sim \mathcal{N}(z; \mathbf{0}, \boldsymbol{I})$$

Then we can train a model to learn a good representation of $p_\theta(x|z)$ with stochastic gradient.

Latent variable models

$$p(x) = \int p_\theta(x|z)p(z)dz$$

Once we have a good representation of $p_\theta(x|z)$, we can sample from $p(x)$ by first sampling

$$z' \sim p(z)$$

and then sampling

$$x' \sim p(x|z')$$

Latent variable models

Problem 1: To learn $p_\theta(x|z)$ using stochastic gradient, we need to know a good mapping

$$f : \mathcal{Z} \times \Theta \mapsto \mathcal{X}$$

In other word when we sample $x \sim p(x|z')$ we need to know which x is likely to be generated by this particular z' in order to train the model.

Latent variable models

Solution: the prior of the latent space can be written has

$$p(z) = \int p(z|x)p(x)dx$$

During training, We can sample z by sampling

$$x' \sim p(x)$$

and then

$$z \sim p(z|x')$$

The training set comes from $p(x)$ so we can sample from it. This will reduce the space of the latent variable a lot and allow the model to learn efficiently.

Latent variable models

Problem 2: $p(z|x)$ is intractable.

Solution: use an approximation $q_\phi(z|x)$

To summarize

- Sample $x \sim D_n$
- Sample $z \sim q_\phi(z|x)$
- Sample $\tilde{x} \sim p_\theta(x|z)$

The parameter to learn are ϕ and θ and they should be learn such that the marginal likelihood $p(x)$ is maximized.

Latent variable models

Before looking at how we can train this model efficiently, let's take a closer look at how it works concretely.

Probabilistic Encoder $q_\phi(z|x)$

Example: Gaussian MLP as encoder

- $\boldsymbol{\mu}_z,\ \log \boldsymbol{\sigma}_z^2 = f(\mathbf{x}; \boldsymbol{\phi})$
  - $\mathbf{h} = \text{relu}\left(xW_{xh} + b_{xh}\right)$
  - $\mu = hW_{hz}^{(1)} + b_{hz}^{(1)}$
  - $\log \sigma^2 = hW_{hz}^{(2)} + b_{hz}^{(2)}$
- $q_\phi(z|x) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_z, \boldsymbol{\sigma}_z^2)$
- $\phi = \{W_{hz}^{(1)}, b_{hz}^{(1)}, W_{hz}^{(2)}, b_{hz}^{(2)}, W_{xh}, b_{xh}\}$

```
class VAE:
    ...
    def encoder(self, x):
        # Encoder network. Return the parameter of q(z|x)
        h = relu(x @ self.Wxh + self.bxh.repeat(x.size(0), 1))
        mu = h @ self.Whz_mu + self.bhz_mu.repeat(x.size(0), 1)
        log_var = h @ self.Whz_var + self.bhz_var.repeat(x.size(0), 1)
        reture mu, log_var
```

Sampling $z \sim q_\phi(z|x)$

Example: Sampling z from $q_\phi(z|x)$

- $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}_z, \boldsymbol{\sigma}_z)$
  - $\boldsymbol{\mu}_z,\ \log \boldsymbol{\sigma}_z^2 = f(\mathbf{x}; \phi)$
  - $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  - $\mathbf{z} = \boldsymbol{\mu}_z + \boldsymbol{\sigma}_z \odot \boldsymbol{\epsilon}$

```
class VAE:
    ...
    def _sample_z(self, mu, log_var):
        epsilon = Variable(torch.randn(mu.size()).to(self._device)
        sigma = torch.exp(log_var / 2)
        return mu + sigma * epsilon
```

Probabilistic Decoder $p_\theta(x|z)$

Example: Gaussian MLP as decoder

- $\boldsymbol{\mu}_x,\ \log \boldsymbol{\sigma}_x^2 = g(\mathbf{x}; \boldsymbol{\theta})$
  - $\mathbf{h} = \text{relu}\left(\mathbf{W}_{zh}\mathbf{x} + \mathbf{b}_{zh}\right)$
  - $\boldsymbol{\mu}_x = W_{hz}^{(1)}\mathbf{h} + \mathbf{b}_{hz}^{(1)}$
  - $\log \boldsymbol{\sigma}_x^2 = W_{hz}^{(2)}\mathbf{h} + \mathbf{b}_{hz}^{(2)}$
- $p_{\boldsymbol{\theta}}(x|z) = \mathcal{N}(x; \boldsymbol{\mu}_x, \log \boldsymbol{\sigma}_x^2)$
- $\boldsymbol{\theta} = \{W_{zh}, b_{zh}, W_{zx}, b_{zx}\}$

```
class VAE:
    ...
    def decoder(self, z):
        # Decoder network. Reconstruct the input from
        # the latent variable z
        h = relu(z @ self.Whx + self.bhx.repeat(x.size(0), 1))
        gamma = h @ self.Whx_mu + self.bhx.repeat(x.size(0), 1)
        reture gamma
```

Latent variable models

To summarize

- Sample $x \sim D_n$
- Sample $z \sim q_\phi(z|x)$
- Sample $\tilde{x} \sim p_\theta(x|z)$

The parameter to learn are ϕ and θ and they should be learn such that the marginal likelihood $p(x)$ is maximized.

Outline

Training VAE

We need $p_\theta(x|z)$ to be such that the marginal likelihood $p(x)$ is maximized.

In other word the lost function should be

$$-\log p(x^{(1)}, ..., x^{(N)}) = -\sum_{i=1}^{N} \log p(x^{(i)})$$

Kullback-Leibler divergence

$$\mathcal{D}_{KL}[q(x)||p(x)] = \mathrm{E}_{x\sim q(x)}[\log q(x) - \log p(x)] \tag{9}$$

Gibbs' inequality:

$$\mathcal{D}_{KL}[q(x)||p(x)] \geq 0 \tag{10}$$

Training VAE

Now let's compute de Kullback-Leibler divergence $\mathcal{D}_{KL}$ between $p(z|x)$ and $q(z|x)$

$$
\begin{aligned}
&\mathcal{D}_{KL}[q(z|x)||p(z|x)] \\
&\qquad =\mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(z|x)] \\
&\qquad =\mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(x|z) - \log p(z) + \log p(x)] \\
&\qquad = \log p(x) + \mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(z)] - \mathrm{E}_{z\sim q(z|x)} \log p(x|z) \\
&\qquad = \log p(x) - \mathcal{D}_{KL}[q(z|x)||p(z|x)] - \mathrm{E}_{z\sim q(z|x)} \log p(x|z)
\end{aligned}
$$

Training VAE

$$\log p(x) = \mathrm{E}_{z\sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)] + \mathcal{D}_{KL}[q(z|x)|| \log p(z|x)] \tag{11}$$

Because of Gibbs' inequality we have

$$\log p(x) \geq \mathrm{E}_{z\sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)]$$

Hence, our loss function is

$$\mathcal{L} = \mathrm{E}_{z\sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)]$$

Solution of $\mathcal{D}_{KL}[q(z|x)||p(z)]$

$$\begin{aligned}\mathcal{D}_{KL}[q_\phi(z|x)||p(z)] &= \int q_\phi(z|x)(\log q_\phi(z|x) - \log p(z))dx \\ &= \int q_\phi(z|x)\log q_\phi(z|x)dx - \int q_\phi(z|x)\log p(z))dx\end{aligned}$$

Training VAE

Suppose $z \in \mathbb{R}^J$ is normal

$$\begin{aligned}\int q(z|x)\log q(z|x)dz &= \int \mathcal{N}(z;\mu,\sigma)\log \mathcal{N}(z;\mu,\sigma) \\ &= -\frac{J}{2}\log 2\pi - \frac{1}{2}\sum_{j=1}^{J}(1+\log \sigma_j^2)\end{aligned} \tag{12}$$

$$\begin{aligned}\int q(z|x)\log p(z)dz &= \int \mathcal{N}(z;\mu,\sigma)\log \mathcal{N}(z;0,I) \\ &= -\frac{J}{2}\log 2\pi - \frac{1}{2}\sum_{j=1}^{J}(\mu_j^2+\sigma_j^2)\end{aligned} \tag{13}$$

Training VAE

$$\mathcal{D}_{KL}[q(z|x)||p(z)] = (12) - (13)$$
$$= \frac{1}{2}\sum_{j=1}^{J} \mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2 \tag{14}$$

```
def kl_divergence(mu, log_sigma):
    sigma = torch.exp(log_sigma)
    return .5 * torch.sum(mu**2 + sigma**2 - 1 - 2*log_sigma, axis=1)
```

VAE Loss Function

$$\begin{aligned}\mathcal{L} &= -\,\mathrm{E}_{z\sim q(z|x)} \log p(x|z) + \mathcal{D}_{KL}[q(z|x)||p(z)] \\ &= -\log \mathcal{N}(x; \boldsymbol{\mu}_x, \boldsymbol{\sigma}_x) + \frac{1}{2}\sum_{j=1}^{J} \mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2\end{aligned} \quad (15)$$

Outline

Unsupervised Spam Detection

- SMS Spam Collecion Data Set[2]
  - Number of instances: 5574
  - Number of spams: 747 ($\sim 13\%$)
- Spam detection
  - Estimate the distribution of all the text messages
  - Evaluate the density of each text message
  - Text message with low density are classified as spam

[2] https://archive.ics.uci.edu/ml/datasets/sms+spam+collection

Data

- Examples of **non spams**
  - *What you doing?how are you?*
  - *Ok lar... Joking wif u oni...*
  - *Cos i was out shopping wif darren jus now n i called him 2 ask wat present he wan lor. Then he started guessing who i was wif n he finally guessed darren lor.*
- Examples of **spams**
  - *FreeMsg: Txt: CALL to No: 86888 & claim your reward of 3 hours talk time to use from your phone now! ubscribe6GBP/ mnth inc 3hrs 16 stop?txtStop*
  - *Sunshine Quiz! Win a super Sony DVD recorder if you canname the capital of Australia? Text MQUIZ to 82277. B*
  - *URGENT! Your Mobile No 07808726822 was awarded a L2,000 Bonus Caller Prize on 02/09/03! This is our 2nd attempt to contact YOU! Call 0871-872-9758 BOX95QUm*

Performance Metric

$$\begin{aligned} \mathrm{F}_1 - \mathrm{Score} &= \left(\frac{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}{2}\right)^{-1} \\ &= 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} \end{aligned} \tag{16}$$

where

$$\mathrm{precision} = \frac{|\{\text{spams detected}\}|}{|\{\text{spams classified}\}|} \qquad \mathrm{recall} = \frac{|\{\text{spams detected}\}|}{|\{\text{spams}\}|}$$

Motivation: Both precision and recall metric are important. Take the harmonic mean between them.

Outline

Preprocessing

- Some terminology
  - Vocabulary: Set of all the word types in the training set $\{w_1, ..., w_{|\text{Vocabulary}|}\}$.
  - Training corpus: Set of all text messages in the training set $\{x^{(1)}, ..., x^{(n)}\}$.
  - Bag of words: Vector representation of a sentence.

Bag of words

- $x^{(i)} =$ *What you doing?how are you?* (original sentence)
- $x^{(i)} =$ *what you do how be you* (after preprocessing)
- $x^{(i)} = [0 \cdots 0\ 1\ 0 \cdots\ 0\ 2\ 0 \cdots 0]$ (vector representation)

```
x = 'What you doing?how are you?'

x = preprocess(x)
print(x)
>>> 'what you do how be you'

x = vectorize(x)
print(x)
>>> [0 0 0 ... 0 0 0]
```

Bag of words

- Bag of words
  - All examples in the corpus have the same length.
  - The value of a position in the vector is equal to the frequency of this word it the example.
  - $x^{(i)} = [0 \cdots 0\ 1\ 0 \cdots\ 0\ 2\ 0 \cdots 0]$
- Binary bag of words
  - All examples in the corpus have the same length.
  - Each element in the vector is either 0 or 1.
  - $x^{(i)} = [0 \cdots 0\ 1\ 0 \cdots\ 0\ 1\ 0 \cdots 0]$

Outline

1. Autoencoders
   - An example to keep in mind
   - Autoencoders
2. Variational autoencoder
   - Generative Model
   - Latent variable models
   - Variational Lower Bound
3. Experiment
   - Unsupervised Spam Detection
   - Preprocessing
   - **Binary Cross Entropy**
   - Bernoulli MLP as Decoder
   - Spam Detector

Binary Cross Entropy

Recall that the density of a Bernoulli distribution is given by

$$\mathrm{Bernoulli}(x;\gamma) = \gamma^x(1-\gamma)^{1-x} \tag{17}$$

where $\gamma \equiv p(x=1)$.

Binary Cross Entropy

Thus the log-likelihood of a Bernoulli distribution is

$$\log p(x) = x \log \gamma + (1 - x) \log(1 - \gamma) \tag{18}$$

Binary Cross Entropy

If your have binary features, i.e. $x_i \in \{0,1\}$ for $i = 1, ..., d$, then the output of the decoder can be interpreted has the parameter of a Bernoulli. Thus the likelihood of the input x can be compute has

$$p(x|z) = \prod_{i=1}^{d} p(x_i|z) = \prod_{i=1}^{d} \gamma_i^{x_i}(1-\gamma_i)^{1-x_i} \tag{19}$$

Note that this works only if $\gamma_i \in (0,1)$. To ensure it, apply sigmoid element wise on the output of the decoder.

Binary Cross Entropy

And the log-likelihood is given by

$$\log p(x|z) = -\sum_{i=1}^{d} x_i \log \gamma_i + (1 - x_i) \log(1 - \gamma_i)$$

```
# in pytorch
functional.binary_cross_entropy(param, x)
```

Outline

Probabilistic Decoder $p_\theta(x|z)$

Bernoulli MLP as decoder

- $\boldsymbol{\gamma} = g(z; \boldsymbol{\theta})$
  - $h = \mathrm{relu}\left(W_{zh}x + b_{zh}\right)$
  - $\gamma = \sigma\left(W_{hx}h + b_{hx}\right)$
- $p_\theta(x|z) = \mathrm{Bernoulli}(x; \boldsymbol{\gamma})$
- $\boldsymbol{\theta} = \{W_{zh}, b_{zh}, W_{zx}, b_{zx}\}$

```
class VAE:
    ...
    def decoder(self, z):
        h = relu(self.W_zh @ z + self.b_zh.repeat(x.size(0), 1))
        gamma = self.W_hx @ h + self.b_hx.repeat(x.size(0), 1)
        reture gamma
```

Outline

Anomaly detection

Algorithm 2 Pseudocode for Batch Gradient Descent

Require: Learning rate ϵ_k
Require: Initial parameter $\boldsymbol{w}_0$
Require: Number of epochs T
  for $i = 1$ to T **do**
    Compute gradient $\boldsymbol{g}_t = \frac{1}{m}\nabla_w \sum_i L(h_{w_{t-1}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \epsilon \boldsymbol{g}_t$
  end for