EE 628
Deep Learning
Fall 2019

Lecture 13

11/21/2019

Assistant Prof. Sergul Aydore

Department of Electrical and Computer Engineering

Overview

- Last lecture we covered
  - Advanced topics in Computer Vision

- Today, we will cover
  - Generative Adversarial Networks
  - Natural Language Processing
  - Q&A about jobs

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.
- Classifiers and regressors are both examples of discriminative learning.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.
- Classifiers and regressors are both examples of discriminative learning.
- But there's more to machine learning than just solving discriminative tasks.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.
- Classifiers and regressors are both examples of discriminative learning.
- But there's more to machine learning than just solving discriminative tasks.
- For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.
- Classifiers and regressors are both examples of discriminative learning.
- But there's more to machine learning than just solving discriminative tasks.
- For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data.
- Given such a model, we could sample synthetic data points that resemble the distribution of the training data.

Generative Adversarial Networks

- Throughout most of this book, we've talked about how to make predictions.
- In some form or another, we used deep neural networks learned mappings from data points to labels.
- This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs.
- Classifiers and regressors are both examples of discriminative learning.
- But there's more to machine learning than just solving discriminative tasks.
- For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data.
- Given such a model, we could sample synthetic data points that resemble the distribution of the training data.
- This kind of learning is called generative modeling.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.
- In statistics, this is called a two-sample test - a test to answer the question whether datasets X and X' were drawn from the same distribution.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.
- In statistics, this is called a two-sample test - a test to answer the question whether datasets X and X' were drawn from the same distribution.
- The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.
- In statistics, this is called a two-sample test - a test to answer the question whether datasets X and X' were drawn from the same distribution.
- The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way.
- In other words, rather than just training a model to say “hey, these two datasets don’t look like they came from the same distribution”, they use the two-sample test to provide training signal to a generative model.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.
- In statistics, this is called a two-sample test - a test to answer the question whether datasets X and X' were drawn from the same distribution.
- The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way.
- In other words, rather than just training a model to say “hey, these two datasets don’t look like they came from the same distribution”, they use the two-sample test to provide training signal to a generative model.
- This allows us to improve the data generator until it generates something that resembles the real data.

Generative Adversarial Networks

- In 2014, a breakthrough paper introduced Generative adversarial networks (GANs), a clever new way to leverage the power of discriminative models to get good generative models.
- At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data.
- In statistics, this is called a two-sample test - a test to answer the question whether datasets X and X' were drawn from the same distribution.
- The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way.
- In other words, rather than just training a model to say “hey, these two datasets don’t look like they came from the same distribution”, they use the two-sample test to provide training signal to a generative model.
- This allows us to improve the data generator until it generates something that resembles the real data.
- At the very least, it needs to fool the classifier.

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.

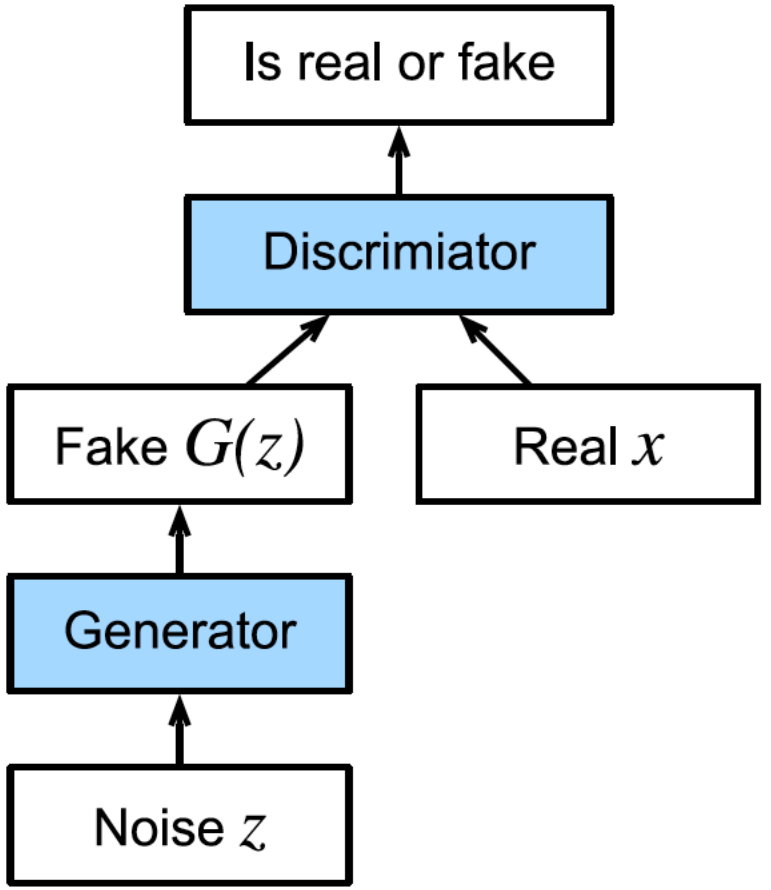


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.
- It attempts to distinguish fake and real data from each other.

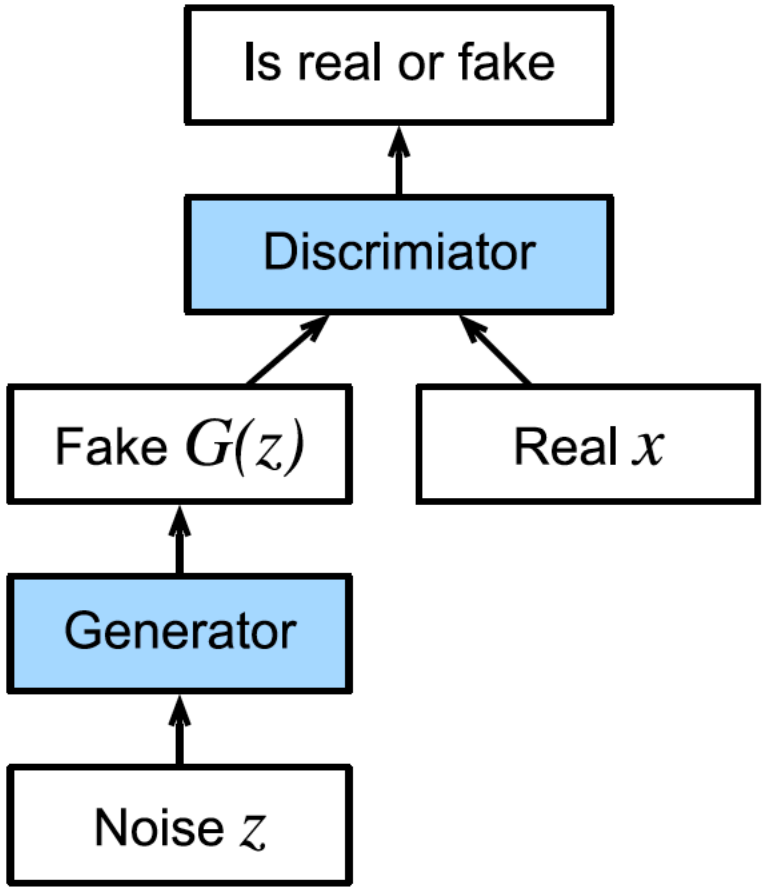


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.
- It attempts to distinguish fake and real data from each other.
- Both networks are in competition with each other.

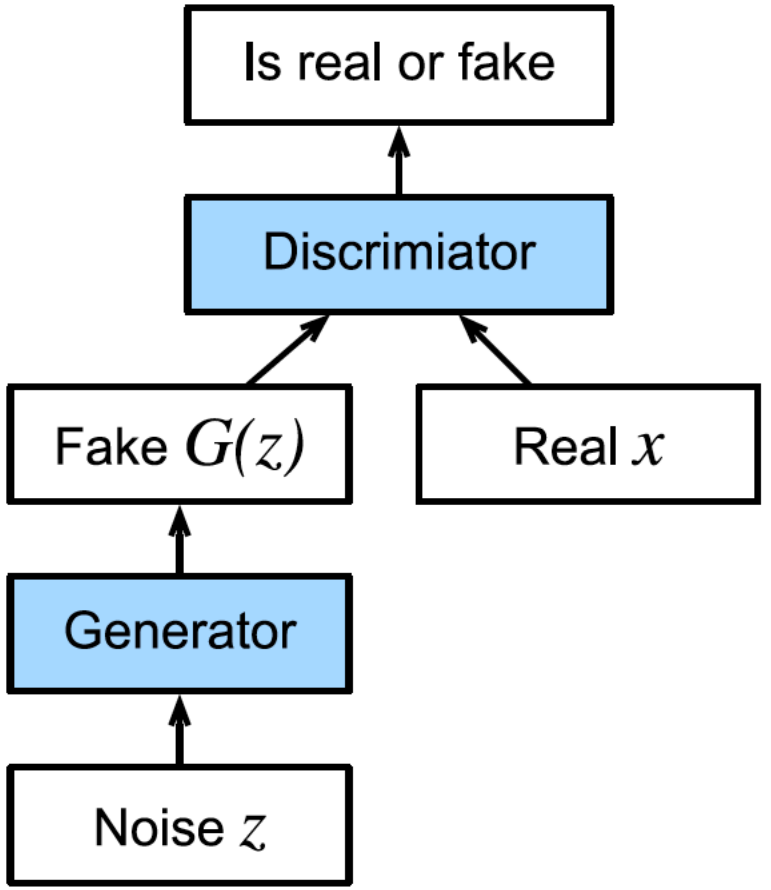


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.
- It attempts to distinguish fake and real data from each other.
- Both networks are in competition with each other.
- The generator network attempts to fool the discriminator network.

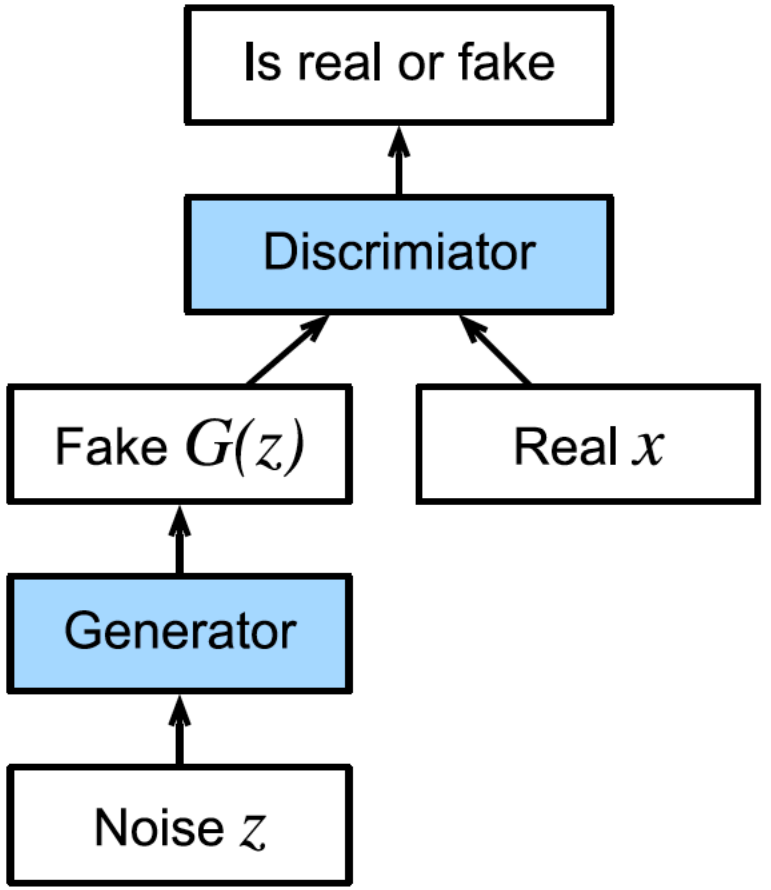


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.
- It attempts to distinguish fake and real data from each other.
- Both networks are in competition with each other.
- The generator network attempts to fool the discriminator network.
- At that point, the discriminator network

adapts to the new fake data.

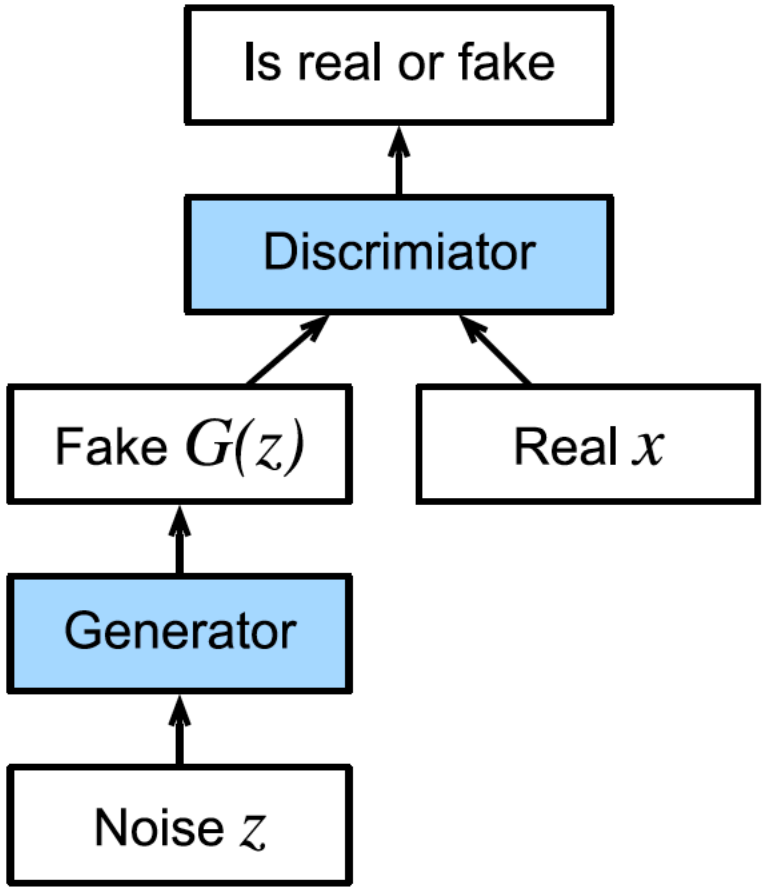


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- There are two pieces to GANs: the generator network and the discriminator network.
- It attempts to distinguish fake and real data from each other.
- Both networks are in competition with each other.
- The generator network attempts to fool the discriminator network.
- At that point, the discriminator network

adapts to the new fake data.

- This information, in turn is used to improve the generator network, and so on.

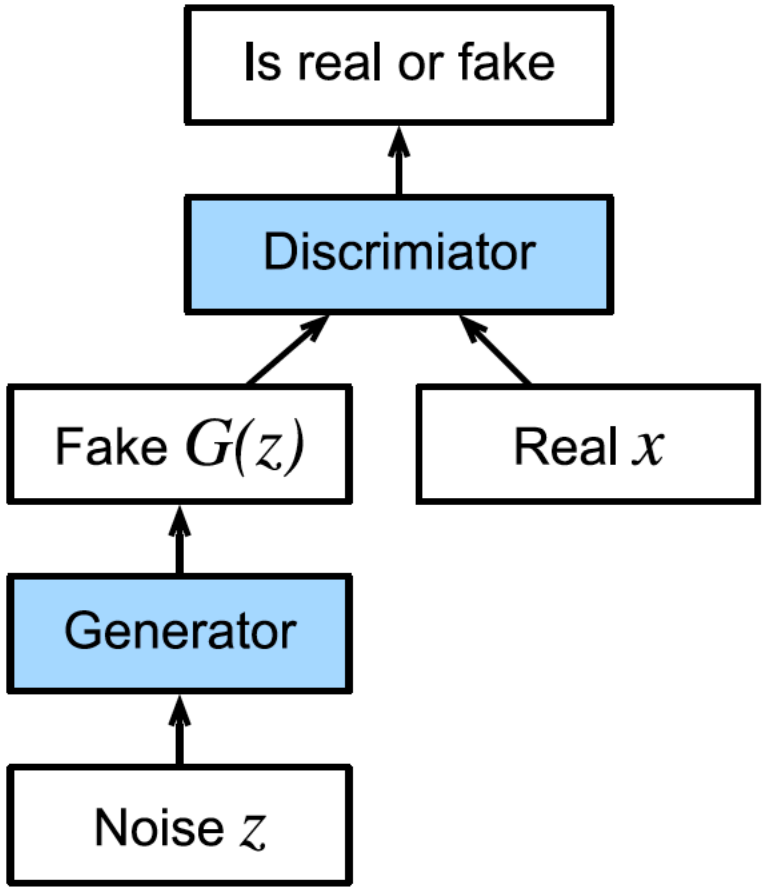


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- The discriminator is a binary classifier to distinguish if the input x is real (from real data) or fake (from the generator).

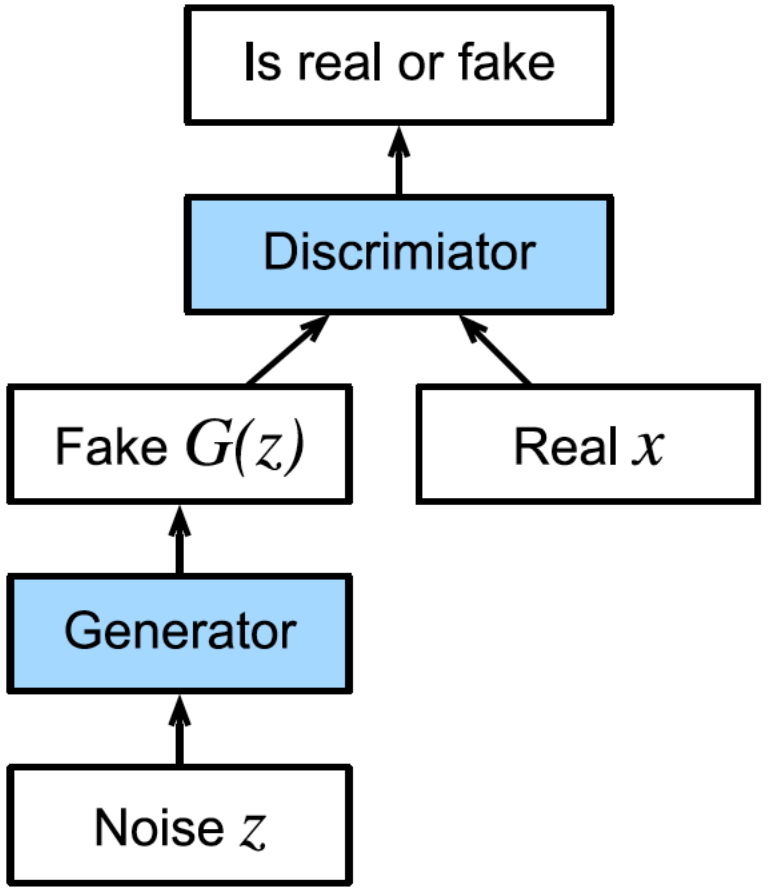


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- The discriminator is a binary classifier to distinguish if the input x is real (from real data) or fake (from the generator).
- Assume the label y for true data is 1 and 0 for fake data.

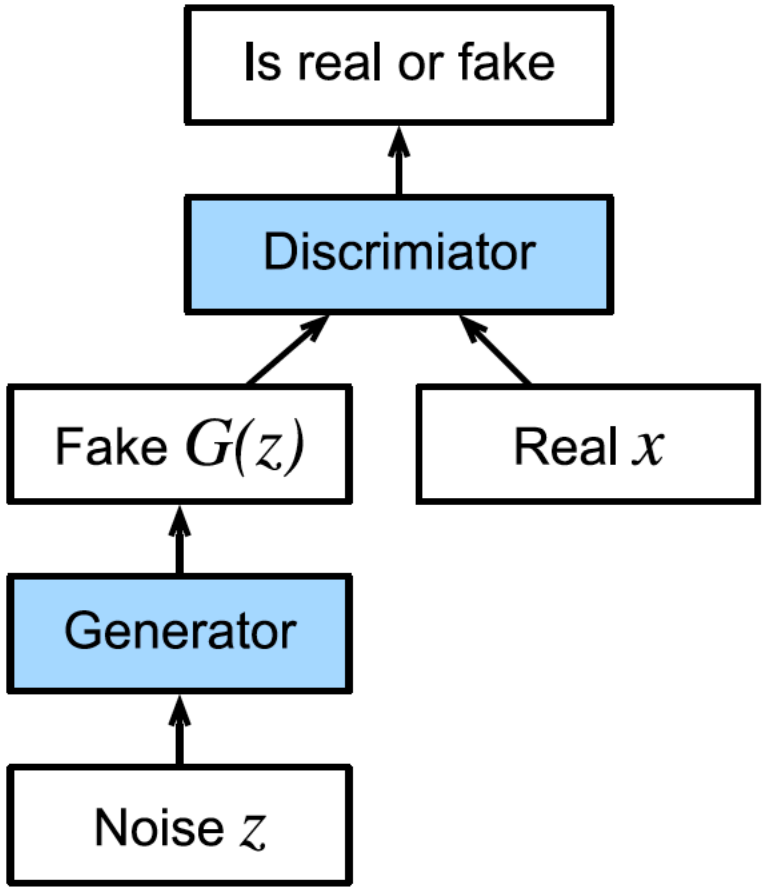


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- The discriminator is a binary classifier to distinguish if the input x is real (from real data) or fake (from the generator).
- Assume the label y for true data is 1 and 0 for fake data.
- We train the discriminator to minimize the cross-entropy loss

$$\min -y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x})),$$

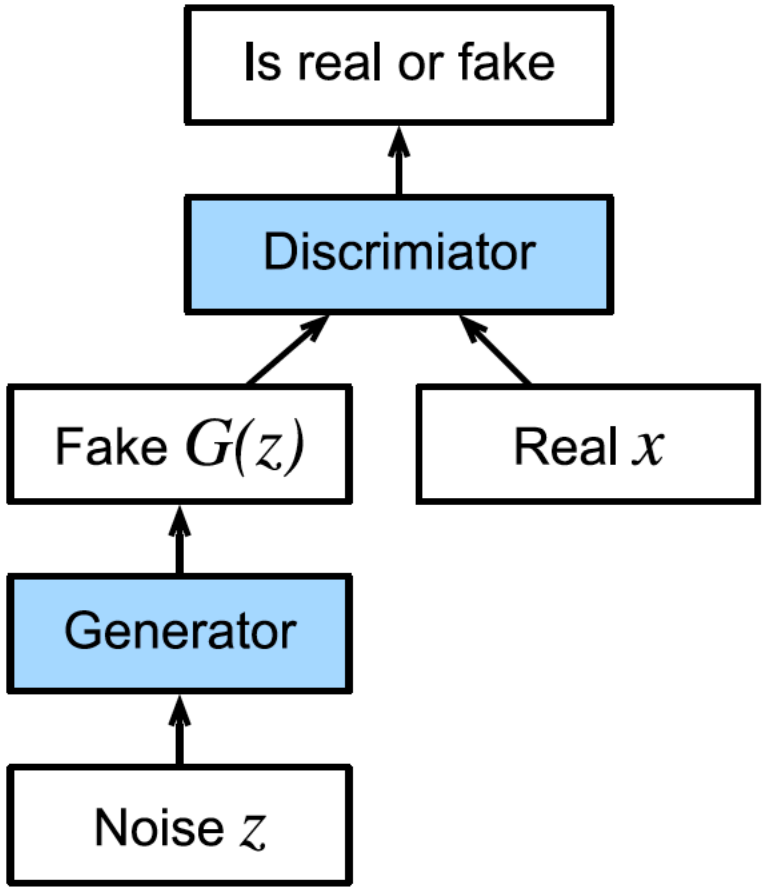


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.

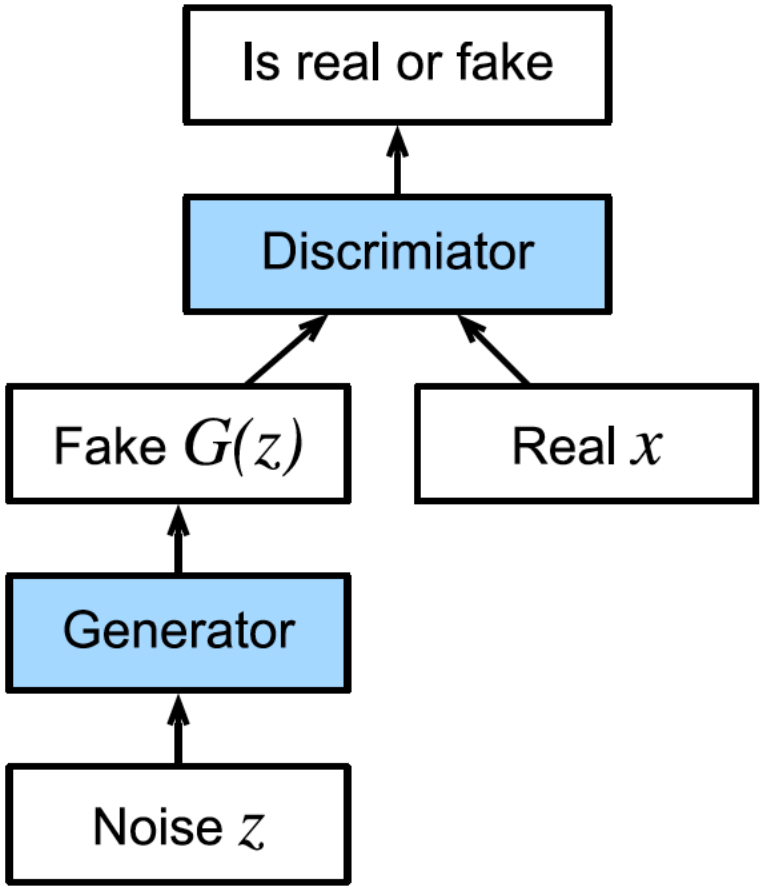


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.

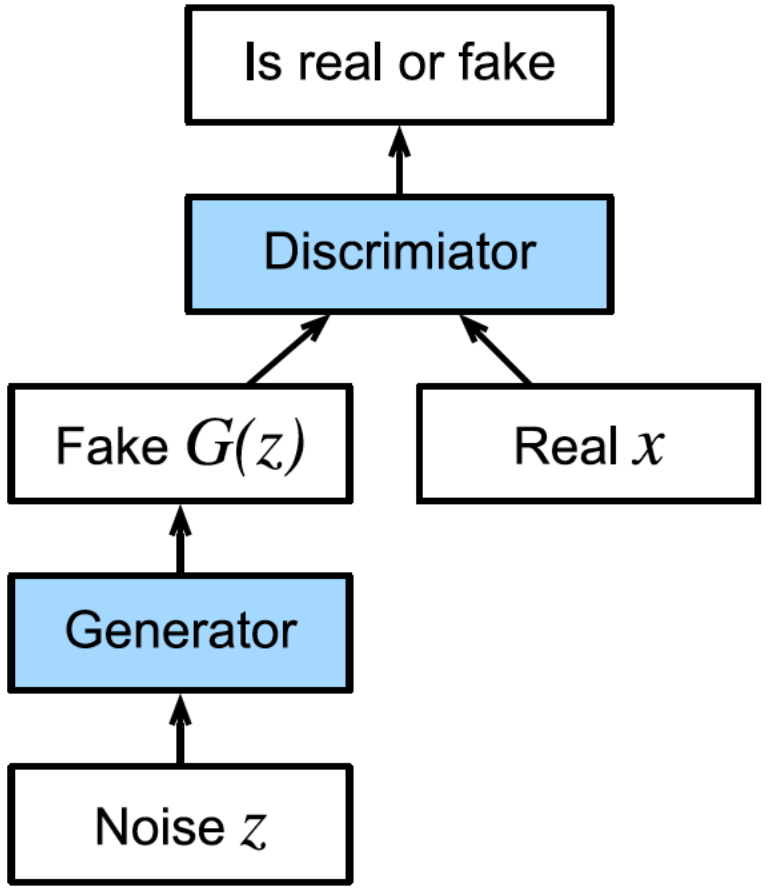


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.
- It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$

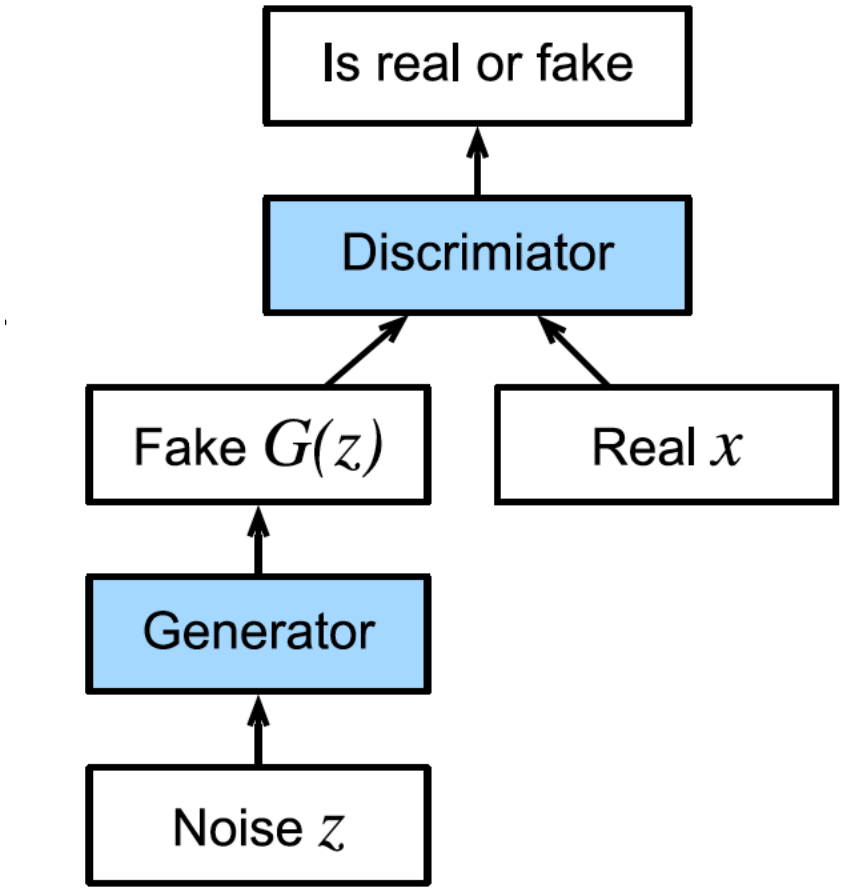


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.
- It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$
- The goal of the generator is to fool the discriminator to classify $\mathbf{x}'$ as true data.

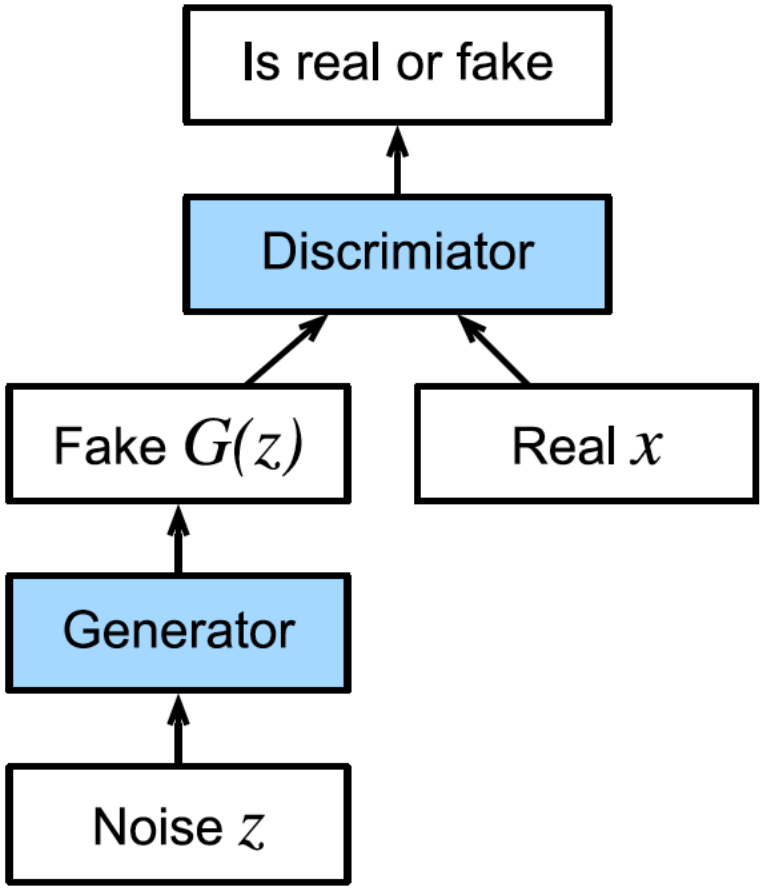


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.
- It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$
- The goal of the generator is to fool the discriminator to classify $\mathbf{x}'$ as true data.
- In other words, we update the parameters of the generator to maximize the cross-entropy loss when $y = 0$.

$$\max -\log(1 - D(\mathbf{x}')).$$

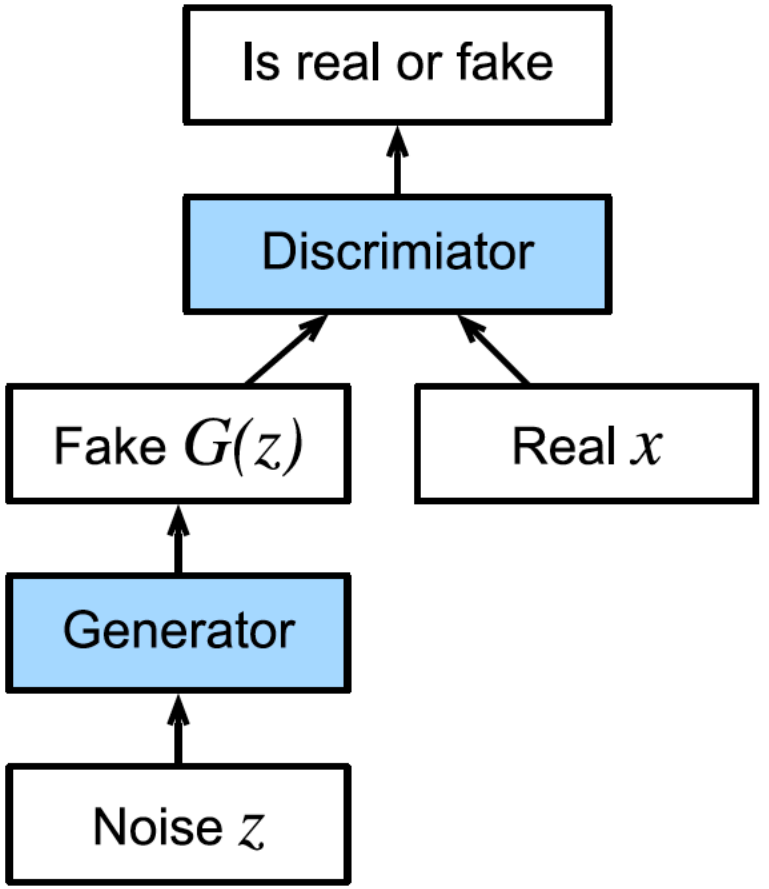


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.
- It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$
- The goal of the generator is to fool the discriminator to classify $\mathbf{x}'$ as true data.
- In other words, we update the parameters of the generator to maximize the cross-entropy loss when $y = 0$.

$$\max -\log(1 - D(\mathbf{x}')).$$

- If the discriminator does a perfect job, then the above loss will be close to 0 which results the gradients are too small to make a good progress for the generator.

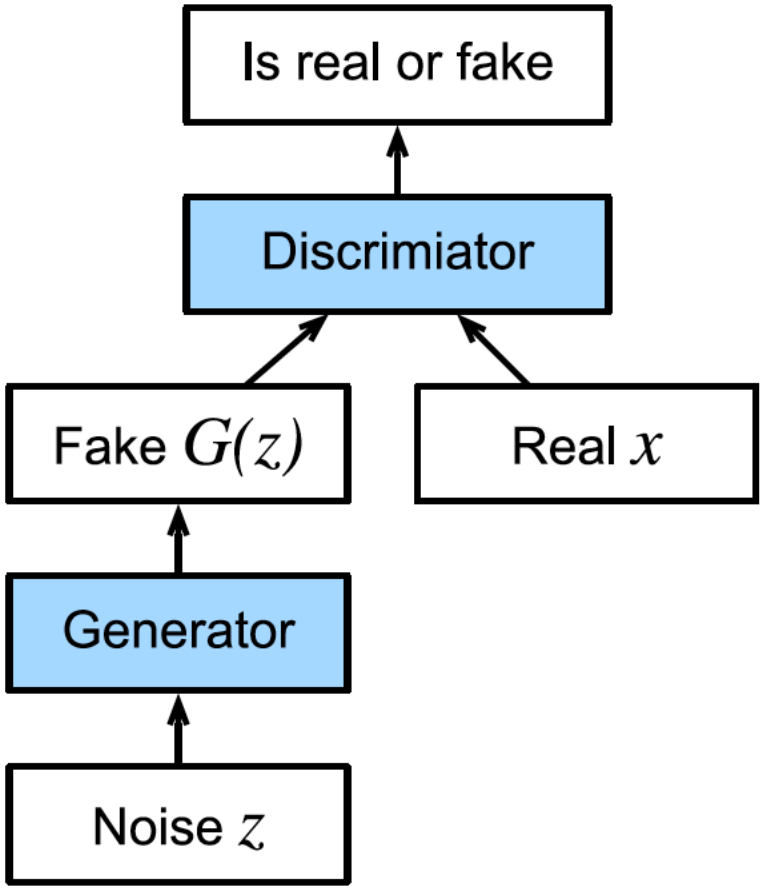


Fig. 16.1.1: Generative Adversarial Networks

Generative Adversarial Networks

- For the generator, it first draws some parameter **z** from a source of randomness.
- We often call **z** the latent variable.
- It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$
- The goal of the generator is to fool the discriminator to classify $\mathbf{x}'$ as true data.
- In other words, we update the parameters of the generator to maximize the cross-entropy loss when $y = 0$.

$$\max -\log(1 - D(\mathbf{x}')).$$

- If the discriminator does a perfect job, then the above loss will be close to 0 which results the gradients are too small to make a good progress for the generator.
- So, commonly, we optimize $\max \log(D(\mathbf{x}'))$,

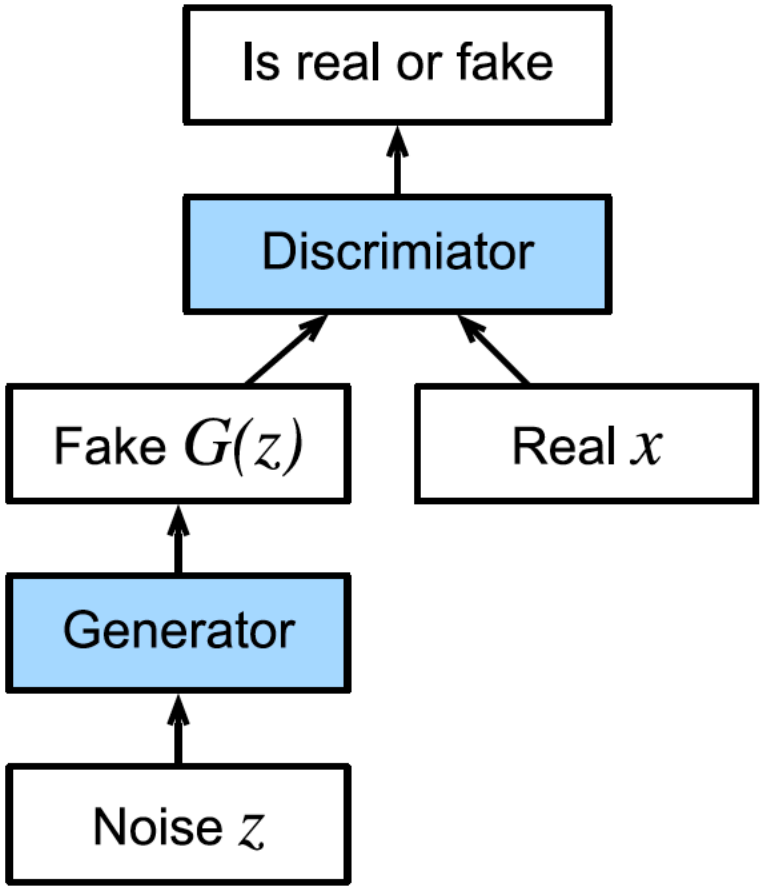


Fig. 16.1.1: Generative Adversarial Networks

Implementing GANs

- Generate some “real” data.

```
X = nd.random.normal(shape=(1000, 2))
A = nd.array([[1, 2], [-0.1, 0.5]])
b = nd.array([1, 2])
data = nd.dot(X, A) + b
```

Implementing GANs

- Generate some “real” data.

```
X = nd.random.normal(shape=(1000, 2))
A = nd.array([[1, 2], [-0.1, 0.5]])
b = nd.array([1, 2])
data = nd.dot(X, A) + b
```

- **Generator**: Our generator network will be the simplest network possible. Hence, it literally only needs to learn the parameters to fake things perfectly.

```
net_G = nn.Sequential()
net_G.add(nn.Dense(2))
```

Implementing GANs

- Generate some “real” data.

```
X = nd.random.normal(shape=(1000, 2))
A = nd.array([[1, 2], [-0.1, 0.5]])
b = nd.array([1, 2])
data = nd.dot(X, A) + b
```

- **Generator**: Our generator network will be the simplest network possible. Hence, it literally only needs to learn the parameters to fake things perfectly.

```
net_G = nn.Sequential()
net_G.add(nn.Dense(2))
```

- **Discriminator**: For the discriminator, we will use an MLP with 3 layers.

```
net_D = nn.Sequential()
net_D.add(nn.Dense(5, activation='tanh'),
          nn.Dense(3, activation='tanh'),
          nn.Dense(1))
```

Preparing Training functions

- First we define a function to update the discriminator.

```
# Save to the d2l package.
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator"""
    batch_size = X.shape[0]
    ones = nd.ones((batch_size,), ctx=X.context)
    zeros = nd.zeros((batch_size,), ctx=X.context)
    with autograd.record():
        real_Y = net_D(X)
        fake_X = net_G(Z)
        # Don't need to compute gradient for net_G, detach it from
        # computing gradients.
        fake_Y = net_D(fake_X.detach())
        loss_D = (loss(real_Y, ones) + loss(fake_Y, zeros)) / 2
    loss_D.backward()
    trainer_D.step(batch_size)
    return loss_D.sum().asscalar()
```

Preparing Training functions

- The generator is updated similarly. Here we reuse the cross-entropy loss but change the label of the fake data from 0 to 1.

```
# Save to the d2l package.
def update_G(Z, net_D, net_G, loss, trainer_G):  # saved in d2l
    """Update generator"""
    batch_size = Z.shape[0]
    ones = nd.ones((batch_size,), ctx=Z.context)
    with autograd.record():
        # We could reuse fake_X from update_D to save computation.
        fake_X = net_G(Z)
        # Recomputing fake_Y is needed since net_D is changed.
        fake_Y = net_D(fake_X)
        loss_G = loss(fake_Y, ones)
    loss_G.backward()
    trainer_G.step(batch_size)
    return loss_G.sum().asscalar()
```

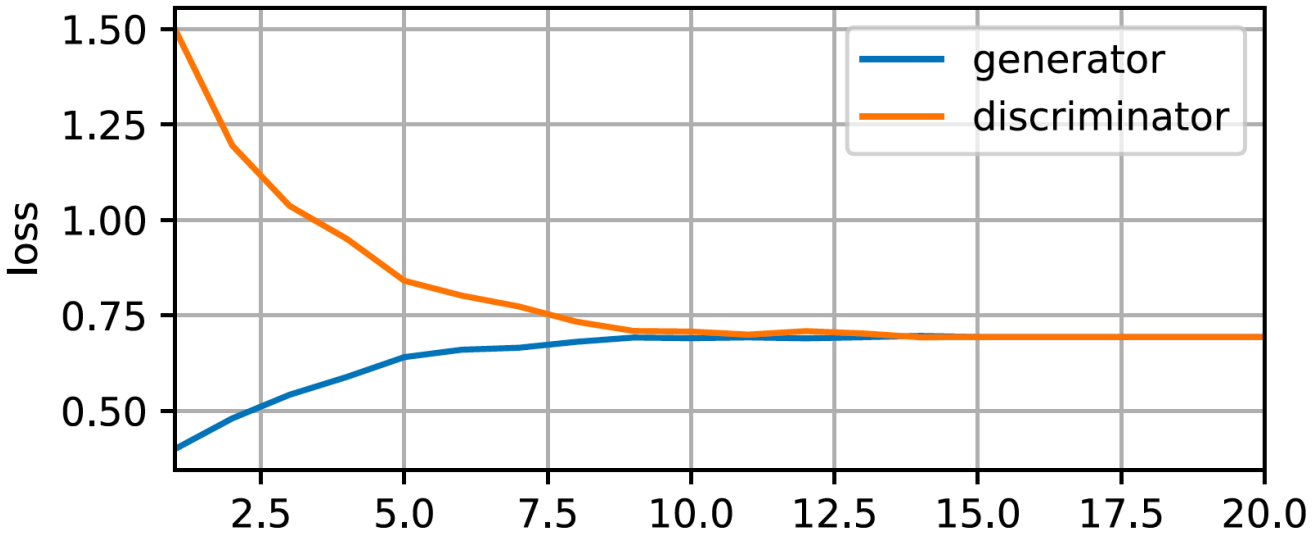
Overall Training

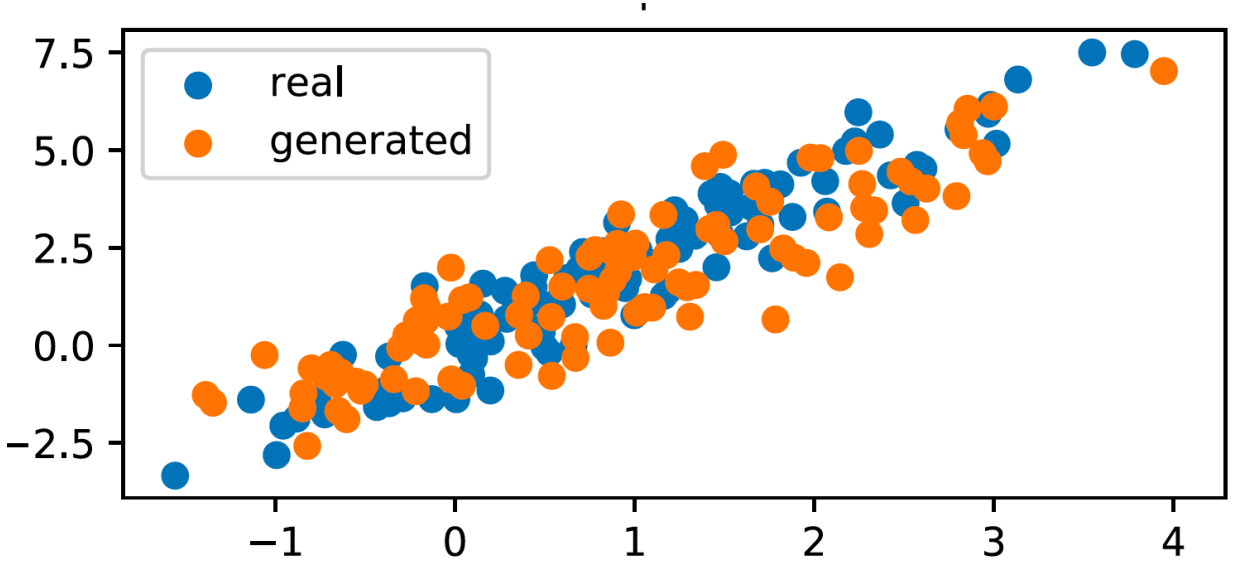
```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True)
    trainer_D = gluon.Trainer(net_D.collect_params(),
                              'adam', {'learning_rate': lr_D})
    for epoch in range(1, num_epochs+1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X in data_iter:
            batch_size = X.shape[0]
            Z = nd.random.normal(0, 1, shape=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Visualize generated examples
        Z = nd.random.normal(0, 1, shape=(100, latent_dim))
        fake_X = net_G(Z).asnumpy()
```

Evaluate Training

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, data[:100].asnumpy())
```

```
loss_D 0.693, loss_G 0.693, 790 examples/sec
```





Deep Convolutional Generative Adversarial Networks

- We introduced the basic ideas behind how GANs work.

Deep Convolutional Generative Adversarial Networks

- We introduced the basic ideas behind how GANs work.
- Now, we'll demonstrate how you can use GANs to generate photorealistic images.

Deep Convolutional Generative Adversarial Networks

- We introduced the basic ideas behind how GANs work.
- Now, we'll demonstrate how you can use GANs to generate photorealistic images.
- We'll use deep convolutional GANs (DCGAN) introduced in https://arxiv.org/pdf/1511.06434.pdf

Pokemon Dataset

Generator

- The generator needs to map the noise variable $\mathbf{z}$, a length-d vector to a RGB image with both width and height to be 64.

Generator

- The generator needs to map the noise variable $\mathbf{z}$, a length-d vector to a RGB image with both width and height to be 64.
- Last week, we introduced the fully convolutional network that uses transposed convolution layer to enlarge input size.

Generator

- The generator needs to map the noise variable $\mathbf{z}$, a length-d vector to a RGB image with both width and height to be 64.
- Last week, we introduced the fully convolutional network that uses transposed convolution layer to enlarge input size.
- The basic block of the generator contains a transposed convolution layer followed by the batch normalization and ReLU activation.

Generator

- The generator needs to map the noise variable $\mathbf{z}$, a length-d vector to a RGB image with both width and height to be 64.
- Last week, we introduced the fully convolutional network that uses transposed convolution layer to enlarge input size.
- The basic block of the generator contains a transposed convolution layer followed by the batch normalization and ReLU activation.

```
class G_block(nn.Block):
    def __init__(self, channels, kernel_size=4,
                 strides=2, padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.Conv2DTranspose(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.Activation('relu')

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

Generator

- The generator consists of four basic blocks that increase input’s both width and height from 1 to 32.

Generator

- The generator consists of four basic blocks that increase input’s both width and height from 1 to 32.
- At the same time, it first projects the latent variable into 64 x 8 channels, and then halve the channels each time.

Generator

- The generator consists of four basic blocks that increase input's both width and height from 1 to 32.
- At the same time, it first projects the latent variable into 64 x 8 channels, and then halve the channels each time.
- At last, a transposed convolution layer is used to generate the output.

Generator

- The generator consists of four basic blocks that increase input's both width and height from 1 to 32.
- At the same time, it first projects the latent variable into 64 x 8 channels, and then halve the channels each time.
- At last, a transposed convolution layer is used to generate the output.
- It further doubles the width and height to match the desired 64 x 64 shape, and reduces the channel size to 3.

Generator

- The generator consists of four basic blocks that increase input's both width and height from 1 to 32.
- At the same time, it first projects the latent variable into 64 x 8 channels, and then halve the channels each time.
- At last, a transposed convolution layer is used to generate the output.
- It further doubles the width and height to match the desired 64 x 64 shape, and reduces the channel size to 3.
- The tanh activation function is applied to project output values into the $(-1, 1)$ range.

Generator

- The generator consists of four basic blocks that increase input's both width and height from 1 to 32.
- At the same time, it first projects the latent variable into 64 x 8 channels, and then halve the channels each time.
- At last, a transposed convolution layer is used to generate the output.
- It further doubles the width and height to match the desired 64 x 64 shape, and reduces the channel size to 3.
- The tanh activation function is applied to project output values into the $(-1, 1)$ range.

```
n_G = 64
net_G = nn.Sequential()
net_G.add(G_block(n_G*8, strides=1, padding=0),  # output: (64*8, 4, 4)
          G_block(n_G*4),  # output: (64*4, 8, 8)
          G_block(n_G*2),  # output: (64*2, 16, 16)
          G_block(n_G),    # output: (64, 32, 32)
          nn.Conv2DTranspose(
              3, kernel_size=4, strides=2, padding=1, use_bias=False,
              activation='tanh'))  # output: (3, 64, 64)
```

Generator

- Generate a 100 dimensional latent variable to verify the generator's output shape.

```
x = nd.zeros((1, 100, 1, 1))
net_G.initialize()
net_G(x).shape
```

```
(1, 3, 64, 64)
```

Discriminator

- The discriminator is a normal convolutional network except that it uses a leaky ReLU as its activation function. Given $\alpha \in [0, 1]$, it is defined as:

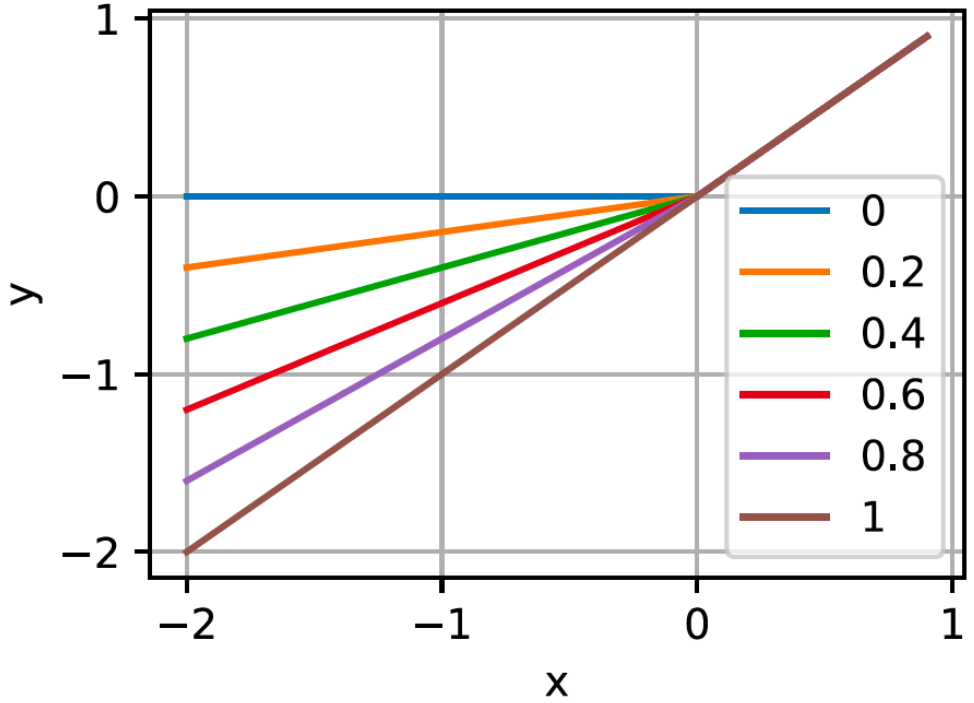
$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}.$$

Discriminator

- The discriminator is a normal convolutional network except that it uses a leaky ReLU as its activation function. Given $\alpha \in [0, 1]$, it is defined as:

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}.$$

- It aims to fix the “dying ReLU” problem that a neuron might always output a negative value and therefore cannot make any progress since the gradient of ReLU is 0.



Discriminator

- The basic block of the discriminator is a convolution layer followed by a batch normalization layer and a leaky ReLU activation.

Discriminator

- The basic block of the discriminator is a convolution layer followed by a batch normalization layer and a leaky ReLU activation.
- The hyper-parameters of the convolution layer are similar to the transpose convolution layer in the generator block.

```
class D_block(nn.Block):
    def __init__(self, channels, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2D(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.LeakyReLU(alpha)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

Discriminator

- The discriminator is a mirror of the generator.

```
n_D = 64
net_D = nn.Sequential()
net_D.add(D_block(n_D),     # output: (64, 32, 32)
          D_block(n_D*2),   # output: (64*2, 16, 16)
          D_block(n_D*4),   # output: (64*4, 8, 8)
          D_block(n_D*8),   # output: (64*8, 4, 4)
          nn.Conv2D(1, kernel_size=4, use_bias=False))  # output: (1, 1, 1)
```

Discriminator

- The discriminator is a mirror of the generator.

```
n_D = 64
net_D = nn.Sequential()
net_D.add(D_block(n_D),     # output: (64, 32, 32)
          D_block(n_D*2),   # output: (64*2, 16, 16)
          D_block(n_D*4),   # output: (64*4, 8, 8)
          D_block(n_D*8),   # output: (64*8, 4, 4)
          nn.Conv2D(1, kernel_size=4, use_bias=False))  # output: (1, 1, 1)
```

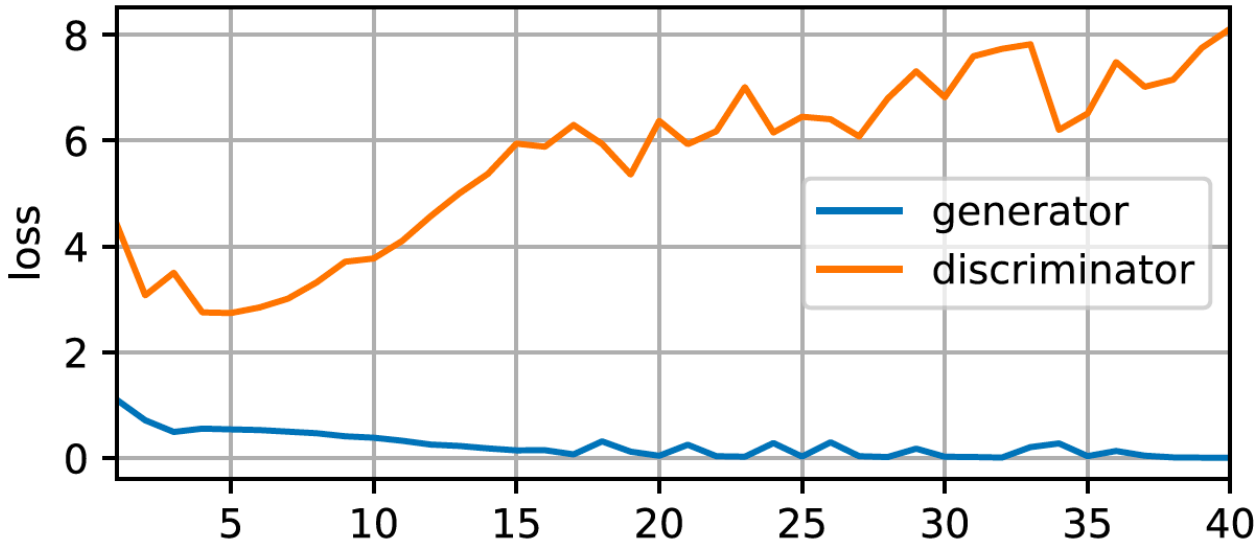
- It uses a convolution layer with output channel 1 as the last layer to obtain a single prediction value.

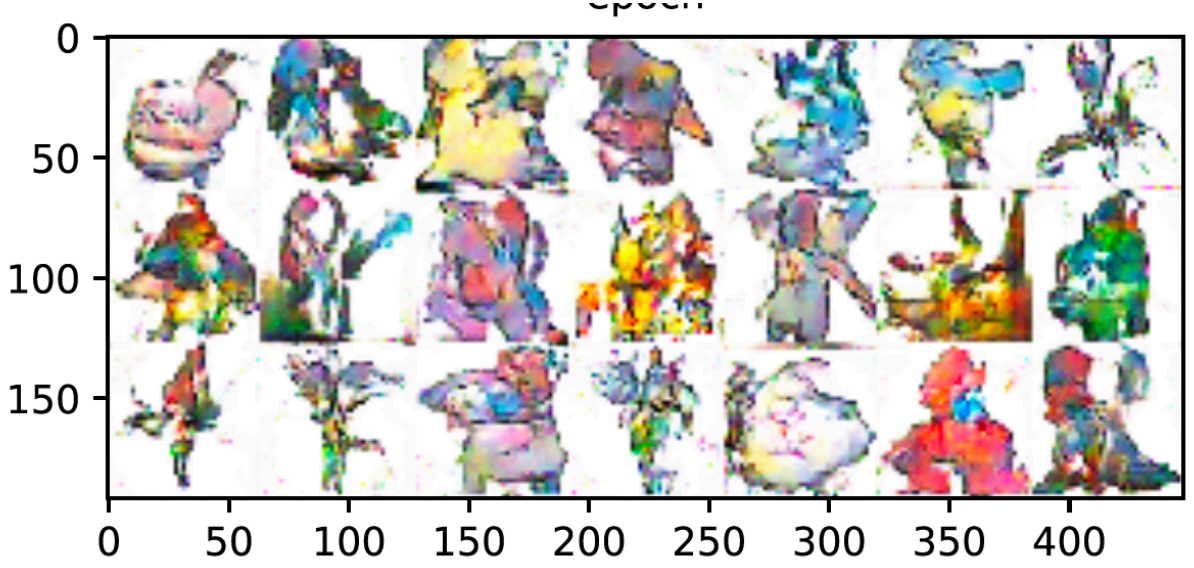
```
x = nd.zeros((1, 3, 64, 64))
net_D.initialize()
net_D(x).shape
```

```
(1, 1, 1, 1)
```

Training

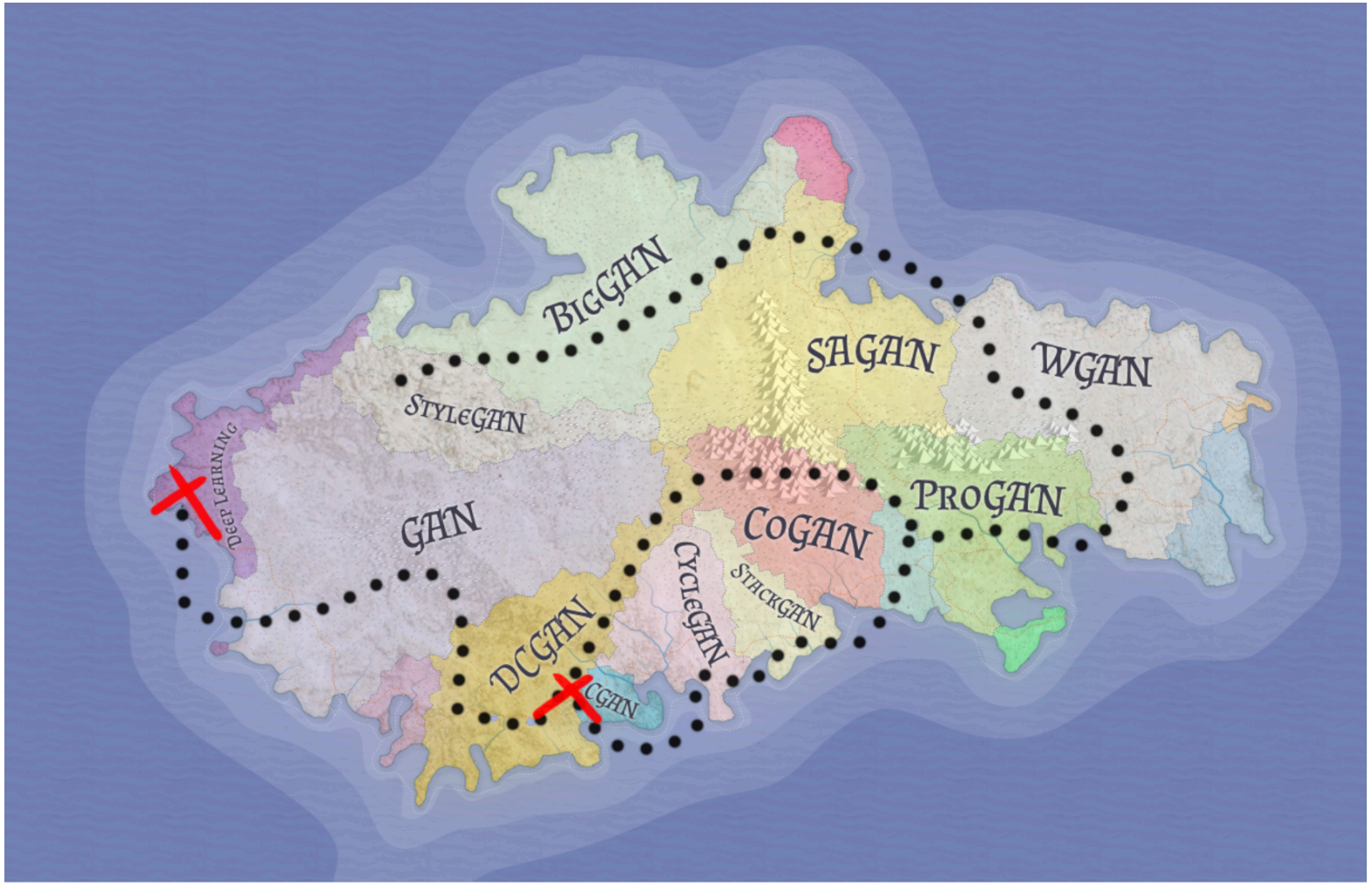
- Training is similar to the basic GAN.





This is just the beginning of the story of GANs...

- You're currently at the second red 'X'



https://blog.floydhub.com/gans-story-so-far/

CGAN: Conditional Generative Adversarial Network

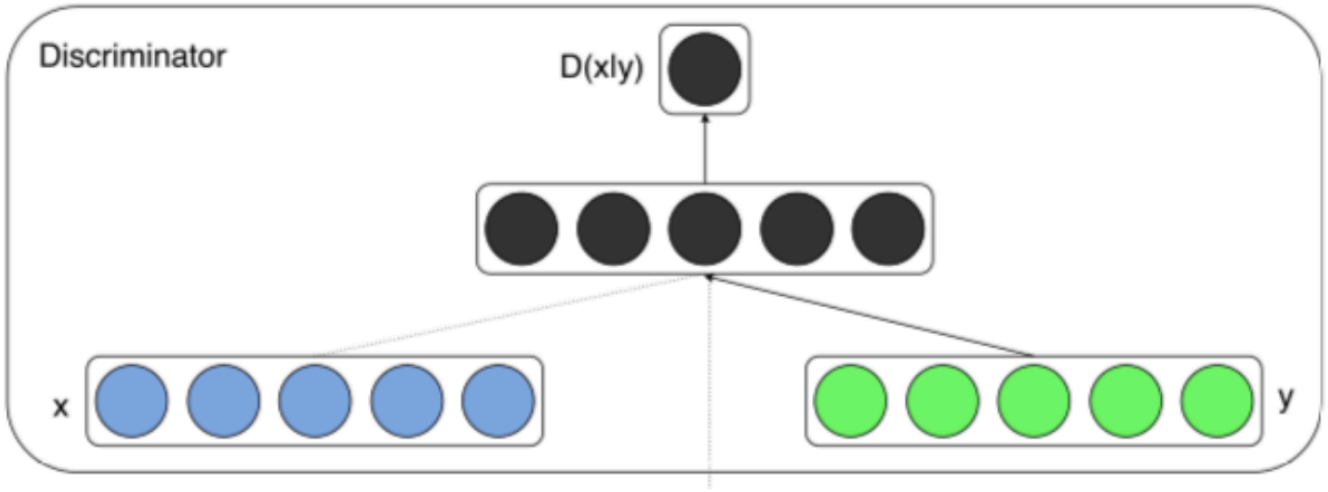
- What if your training dataset includes both cats and dogs? Your GAN will produce blurry half breeds.

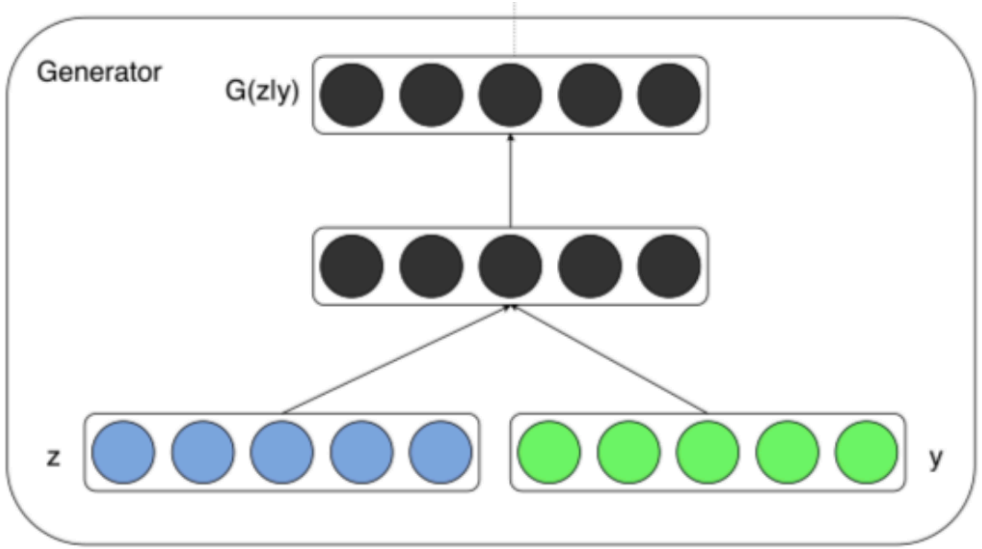
CGAN: Conditional Generative Adversarial Network

- What if your training dataset includes both cats and dogs? Your GAN will produce blurry half breeds.
- CGAN aims to solve this issue by telling the generator to generate images of only one particular class, like a cat, dog, or Nicholas Cage.

CGAN: Conditional Generative Adversarial Network

- What if your training dataset includes both cats and dogs? Your GAN will produce blurry half breeds.
- CGAN aims to solve this issue by telling the generator to generate images of only one particular class, like a cat, dog, or Nicholas Cage.
- CGAN concatenates a one-hot vector y to the random noise vector z to result in an architecture:





https://arxiv.org/pdf/1411.1784.pdf

CycleGAN (https://arxiv.org/abs/1703.10593v6)

- GANs aren't used just for generating images. They can also be used to learn a mapping from one domain of images to another. So we train it on say, the set of all Monet paintings.

ProGAN: Progressive growing of Generative Adversarial Networks (https://arxiv.org/pdf/1710.10196.pdf)

- There are many problems with training GANs. The most important of which is the training instability.

ProGAN: Progressive growing of Generative Adversarial Networks (https://arxiv.org/pdf/1710.10196.pdf)

- There are many problems with training GANs. The most important of which is the training instability.
- ProGAN is a technique that helps stabilize GAN training by incrementally increasing the resolution of the generated image.

ProGAN: Progressive growing of Generative Adversarial Networks (https://arxiv.org/pdf/1710.10196.pdf)

- There are many problems with training GANs. The most important of which is the training instability.
- ProGAN is a technique that helps stabilize GAN training by incrementally increasing the resolution of the generated image.
- The intuition here is that it’s easier to generate a 4x4 image than it is to generate a 1024x1024 image. Also, it’s easier to map a 16x16 image to a 32x32 image than it is to map a 2x2 image to a 32x32 image.

ProGAN: Progressive growing of Generative Adversarial Networks (https://arxiv.org/pdf/1710.10196.pdf)

- There are many problems with training GANs. The most important of which is the training instability.
- ProGAN is a technique that helps stabilize GAN training by incrementally increasing the resolution of the generated image.
- The intuition here is that it’s easier to generate a 4x4 image than it is to generate a 1024x1024 image. Also, it’s easier to map a 16x16 image to a 32x32 image than it is to map a 2x2 image to a 32x32 image.
- So ProGAN first trains a 4x4 generator and a 4x4 discriminator and adds layers that correspond to higher resolutions later in the training process.

ProGAN: Progressive growing of Generative Adversarial Networks (https://arxiv.org/pdf/1710.10196.pdf)

- There are many problems with training GANs. The most important of which is the training instability.
- ProGAN is a technique that helps stabilize GAN training by incrementally increasing the resolution of the generated image.
- The intuition here is that it's easier to generate a 4x4 image than it is to generate a 1024x1024 image. Also, it's easier to map a 16x16 image to a 32x32 image than it is to map a 2x2 image to a 32x32 image.
- So ProGAN first trains a 4x4 generator and a 4x4 discriminator and adds layers that correspond to higher resolutions later in the training process.
- https://cdn-images-1.medium.com/max/1600/1*tUhgr3m54Qc80GU2BkaOiQ.gif

WGAN: Wasserstein Generative Adversarial Networks https://arxiv.org/abs/1701.07875v3

- In short, WGAN (the 'W' stands for Wasserstein) proposes a new cost function that some nice properties
- The original GAN paper showed that when the discriminator is optimal, the generator is updated in such a way to minimize the Jensen-Shannon divergence.

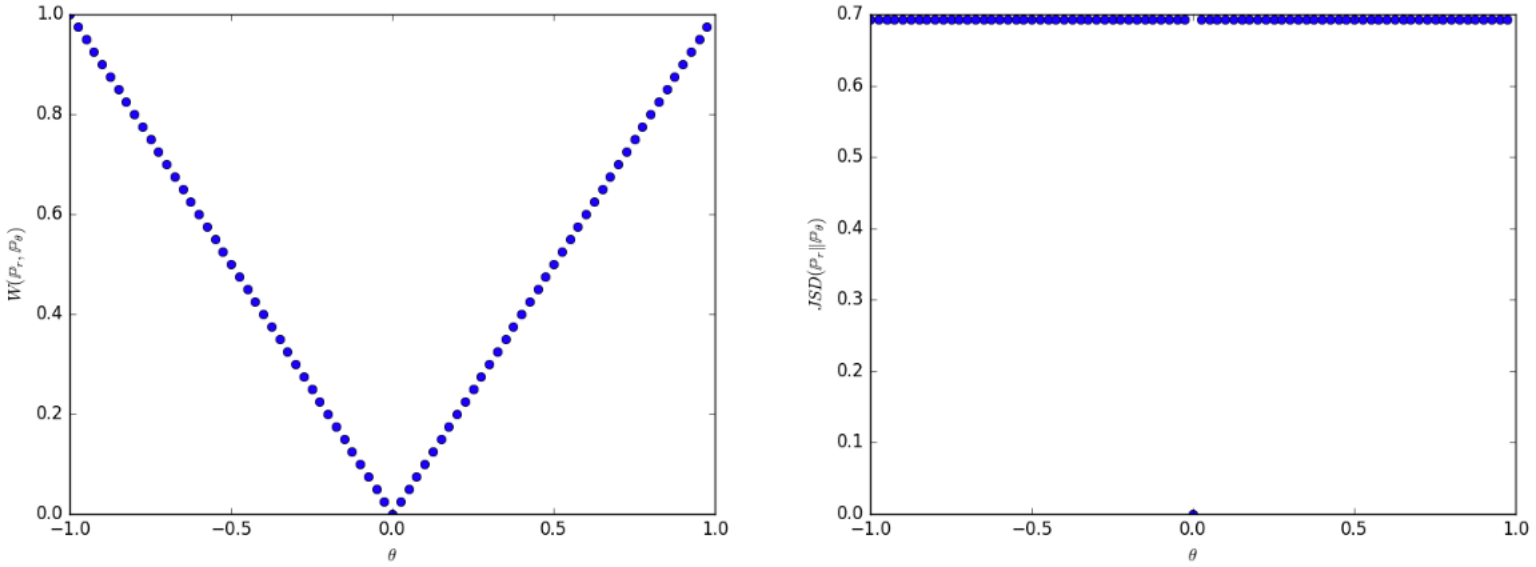


Figure 1: These plots show $\rho(\mathbb{P}_\theta, \mathbb{P}_0)$ as a function of θ when ρ is the EM distance (left plot) or the JS divergence (right plot). The EM plot is continuous and provides a usable gradient everywhere. The JS plot is not continuous and does not provide a usable gradient.

NATURAL LANGUAGE PROCESSING

- Now, we will discuss how to use vectors to represent words and train the word vectors on a corpus.

NATURAL LANGUAGE PROCESSING

- Now, we will discuss how to use vectors to represent words and train the word vectors on a corpus.
- We will also use word vectors pre-trained on a larger corpus to find synonyms and analogies.

NATURAL LANGUAGE PROCESSING

- Now, we will discuss how to use vectors to represent words and train the word vectors on a corpus.
- We will also use word vectors pre-trained on a larger corpus to find synonyms and analogies.
- Then, in the text classification task, we will use word vectors to analyze the emotion of a text and explain the important ideas on RNNs and the convolutional neural networks.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.
- Words are commonly used as the unit of analysis in natural language processing.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.
- Words are commonly used as the unit of analysis in natural language processing.
- A word vector is a vector used to represent a word.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.
- Words are commonly used as the unit of analysis in natural language processing.
- A word vector is a vector used to represent a word.
- The technique of mapping words to vectors of real numbers is also known as word embedding.

Word Embedding (word2vec)

- A natural language is a complex system that we use to communicate.
- Words are commonly used as the unit of analysis in natural language processing.
- A word vector is a vector used to represent a word.
- The technique of mapping words to vectors of real numbers is also known as word embedding.
- Over the last few years, word embedding has gradually become basic knowledge in natural language processing.

Why not use One-hot vectors?

- We used one-hot vectors to represent words in our text processing examples.

Why not use One-hot vectors?

- We used one-hot vectors to represent words in our text processing examples.
- Each word is represented as a vector of length N (dictionary size) that can be used directly by the neural network.

Why not use One-hot vectors?

- We used one-hot vectors to represent words in our text processing examples.
- Each word is represented as a vector of length N (dictionary size) that can be used directly by the neural network.
- Although one-hot word vectors are easy to construct, they are usually not a good choice.

Why not use One-hot vectors?

- We used one-hot vectors to represent words in our text processing examples.
- Each word is represented as a vector of length N (dictionary size) that can be used directly by the neural network.
- Although one-hot word vectors are easy to construct, they are usually not a good choice.
- The reason is that the one-hot word vectors cannot accurately express the similarity between different words.

Cosine Similarity

- For the vectors $\mathbf{x}, \mathbf{y} \in R^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [-1, 1].$$

Cosine Similarity

- For the vectors $\mathbf{x}, \mathbf{y} \in R^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [-1, 1].$$

- Since the cosine similarity between the one-hot vectors of any two different words is 0, it is difficult to use the one-hot vector to accurately represent the similarity between multiple different words.

The Skip-Gram Model

- The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence.

The Skip-Gram Model

- The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence.
- For example, we assume that the text sequence is “the”, “man”, “loves”, “his”, and “son”. We use “loves” as the central target word and set the context window size to 2.

The Skip-Gram Model

- The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence.
- For example, we assume that the text sequence is “the”, “man”, “loves”, “his”, and “son”. We use “loves” as the central target word and set the context window size to 2.
- The skip-gram model is concerned with the conditional probability for generating the context words, “the”, “man”, “his” and “son”, that are within a distance of no more than 2 words

$$\mathbb{P}(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}).$$

The Skip-Gram Model

- The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence.
- For example, we assume that the text sequence is “the”, “man”, “loves”, “his”, and “son”. We use “loves” as the central target word and set the context window size to 2.
- The skip-gram model is concerned with the conditional probability for generating the context words, “the”, “man”, “his” and “son”, that are within a distance of no more than 2 words

$$\mathbb{P}(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}).$$

- We assume that, given the central target word, the context words are generated independently of each other.

$$\mathbb{P}(\text{"the"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"man"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"his"} \mid \text{"loves"}) \cdot \mathbb{P}(\text{"son"} \mid \text{"loves"}).$$

The Skip-Gram Model

- In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability.

The Skip-Gram Model

- In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability.
- We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in R^d$ when it is the central target word, and $\mathbf{u}_i \in R^d$ when it is a context word.

The Skip-Gram Model

- In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability.
- We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in R^d$ when it is the central target word, and $\mathbf{u}_i \in R^d$ when it is a context word.
- Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary.

The Skip-Gram Model

- In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability.
- We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in R^d$ when it is the central target word, and $\mathbf{u}_i \in R^d$ when it is a context word.
- Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary.
- The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product:

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)},$$

The Skip-Gram Model

- Assume that a text sequence of length T is given, where the word at time step t is denoted as $w^{(t)}$.

The Skip-Gram Model

- Assume that a text sequence of length T is given, where the word at time step t is denoted as $w^{(t)}$.
- Assume that context words are independently generated given center words.

The Skip-Gram Model

- Assume that a text sequence of length T is given, where the word at time step t is denoted as $w^{(t)}$.
- Assume that context words are independently generated given center words.
- When context window size is m, the likelihood function of the skip-gram model is the joint probability of generating all the context words given any center word

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}),$$

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.

T

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$.

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$.
- If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence.

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$.
- If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence.
- By definition, we have $\log \mathbb{P}(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$.
- If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence.
- By definition, we have $\log \mathbb{P}(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$
- After the training, for any word in the dictionary with index i, we are going to get its two word vector sets $\mathbf{v}_i$ and $\mathbf{u}_i$.

Skip-Gram Model Training

- The skip-gram model parameters are the central target word vector and context word vector for each individual word.
- In the training process, we are going to learn the model parameters by maximizing the likelihood function.
- This is equivalent to minimizing the following loss function: $-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$.
- If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence.
- By definition, we have $\log \mathbb{P}(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$
- After the training, for any word in the dictionary with index i, we are going to get its two word vector sets $\mathbf{v}_i$ and $\mathbf{u}_i$.
- In applications of natural language processing (NLP), the central target word vector in the skip-gram model is generally used as the representation vector of a word.

The Continuous Bag of Words (CBOW) Model

- The continuous bag of words (CBOW) model is similar to the skip-gram model.

The Continuous Bag of Words (CBOW) Model

- The continuous bag of words (CBOW) model is similar to the skip-gram model.
- The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence.

The Continuous Bag of Words (CBOW) Model

- The continuous bag of words (CBOW) model is similar to the skip-gram model.
- The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence.
- With the same text sequence “the”, “man”, “loves”, “his” and “son”, in which “loves” is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word “loves” based on the context words “the”, “man”, “his” and “son”, such as

$$\mathbb{P}(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}).$$

The Continuous Bag of Words (CBOW) Model

- The continuous bag of words (CBOW) model is similar to the skip-gram model.
- The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence.
- With the same text sequence “the”, “man”, “loves”, “his” and “son”, in which “loves” is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word “loves” based on the context words “the”, “man”, “his” and “son”, such as

$$\mathbb{P}(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}).$$

- Since there are multiple context words in the CBOW model, we will average their word vectors and then use the same method as the skip-gram model to compute the conditional probability.

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.
- Let central target vector w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexes as $o_1, \ldots, o_{2m}$ in the dictionary.

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.
- Let central target vector w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexes as $o_1, \ldots, o_{2m}$ in the dictionary.
- Thus, the conditional probability of generating a central target word from the given context word is

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}.$$

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.
- Let central target vector w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexes as $o_1, \ldots, o_{2m}$ in the dictionary.
- Thus, the conditional probability of generating a central target word from the given context word is

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}.$$

- For brevity, denote $W_0 = \{w_{o_1}, \ldots, w_{o_{2m}}\}$, and $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1}, \ldots, \mathbf{v}_{o_{2m}})/2m$.

The Continuous Bag of Words (CBOW) Model

- We assume that $\mathbf{v}_i \in R^d$ and $\mathbf{u}_i \in R^d$ are the context word vector and central target word vector of the word with index i in the dictionary.
- Let central target vector w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexes as $o_1, \ldots, o_{2m}$ in the dictionary.
- Thus, the conditional probability of generating a central target word from the given context word is

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}.$$

- For brevity, denote $W_0 = \{w_{o_1}, \ldots, w_{o_{2m}}\}$, and $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1}, \ldots, \mathbf{v}_{o_{2m}})/2m$.
- The conditional probability equation can be simplified as:

$$\mathbb{P}(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}.$$

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- Notice that $\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp \left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right)$

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- Notice that $\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp \left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right)$
- We can compute the gradient of any context word vector v_{o_i} $(i = 1, \ldots, 2m)$.

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- Notice that $\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp \left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right)$
- We can compute the gradient of any context word vector v_{o_i} $(i = 1, \ldots, 2m)$.
- We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model,

CBOW Model Training

- Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m.
- The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

- Notice that $\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp \left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right)$
- We can compute the gradient of any context word vector v_{o_i} $(i = 1, \ldots, 2m)$.
- We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model,
- We usually use the context word vector as the representation vector for a word in the CBOW model.

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right)$$

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- The loss above includes the sum of the number of items in the dictionary size.

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- The loss above includes the sum of the number of items in the dictionary size.
- The gradient computation for each step contains the sum of the number of items in the dictionary size.

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- The loss above includes the sum of the number of items in the dictionary size.
- The gradient computation for each step contains the sum of the number of items in the dictionary size.
- For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high.

Approximate Training for Word2vec

- The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

- The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log\left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)\right)$$

- The loss above includes the sum of the number of items in the dictionary size.
- The gradient computation for each step contains the sum of the number of items in the dictionary size.
- For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high.
- In order to reduce such computational complexity, we will introduce two approximate training methods in this section: **negative sampling** and **hierarchical softmax**.

Negative Sampling

- Negative sampling modifies the original objective function.

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$
- We will first consider training the word vector by maximizing the joint probability of all events in the text sequence.

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$
- We will first consider training the word vector by maximizing the joint probability of all events in the text sequence.
- We consider maximizing the joint probability $\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \neq 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)})$

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$
- We will first consider training the word vector by maximizing the joint probability of all events in the text sequence.
- We consider maximizing the joint probability $\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)})$
- However, the events included in the model only consider positive examples.

Negative Sampling

- Negative sampling modifies the original objective function.
- Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from $\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$
- We will first consider training the word vector by maximizing the joint probability of all events in the text sequence.
- We consider maximizing the joint probability $\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)})$
- However, the events included in the model only consider positive examples.
- Negative sampling makes the objective function more meaningful by sampling with an addition of negative examples.

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.
- By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.
- By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

- Here, the conditional probability is approximated to be

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.
- By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

- Here, the conditional probability is approximated to be

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

- The logarithmic loss for the conditional probability above is

$$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \mathbb{P}(D = 0 \mid w^{(t)}, w_k)$$

Negative Sampling

- We sample K words that do not appear in the context window to act as noise words.
- By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \le j \le m,\ j \ne 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

- Here, the conditional probability is approximated to be

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

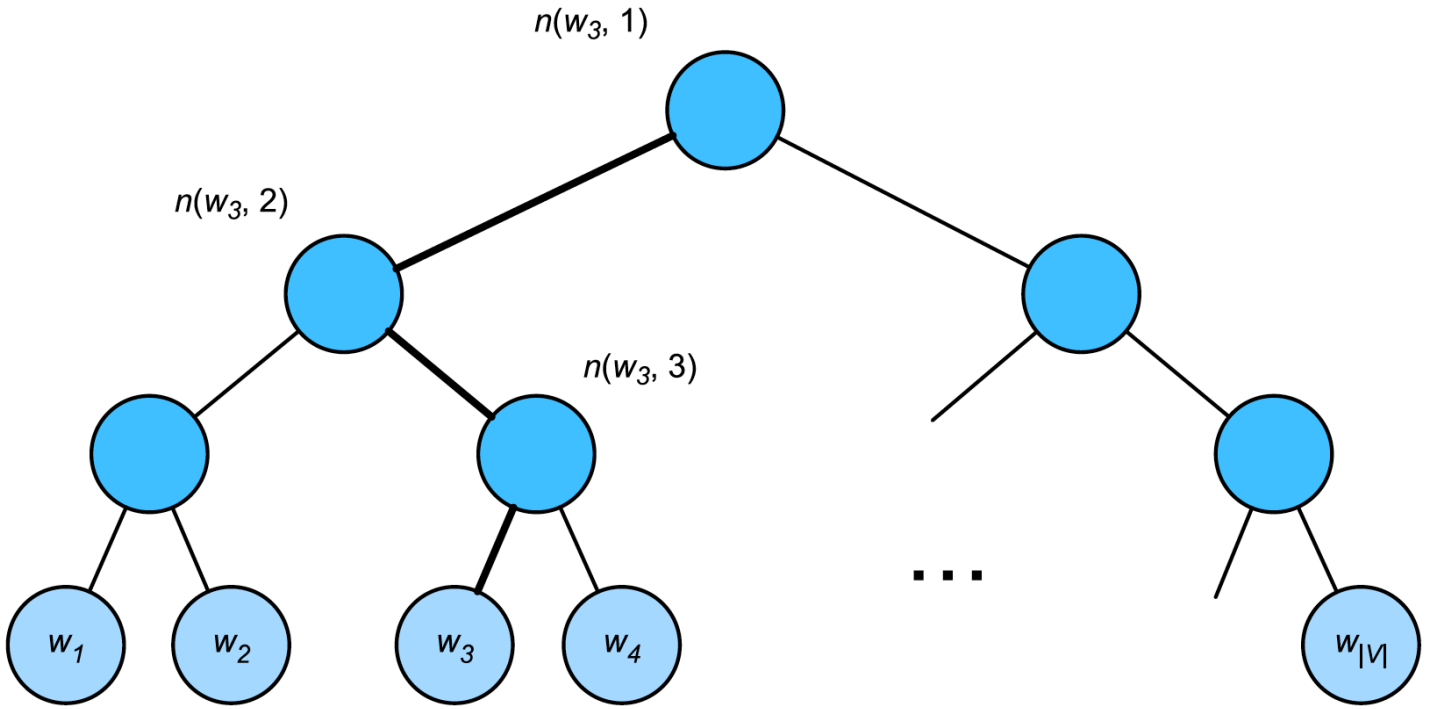
- The logarithmic loss for the conditional probability above is

$$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \mathbb{P}(D = 0 \mid w^{(t)}, w_k)$$

- Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K.

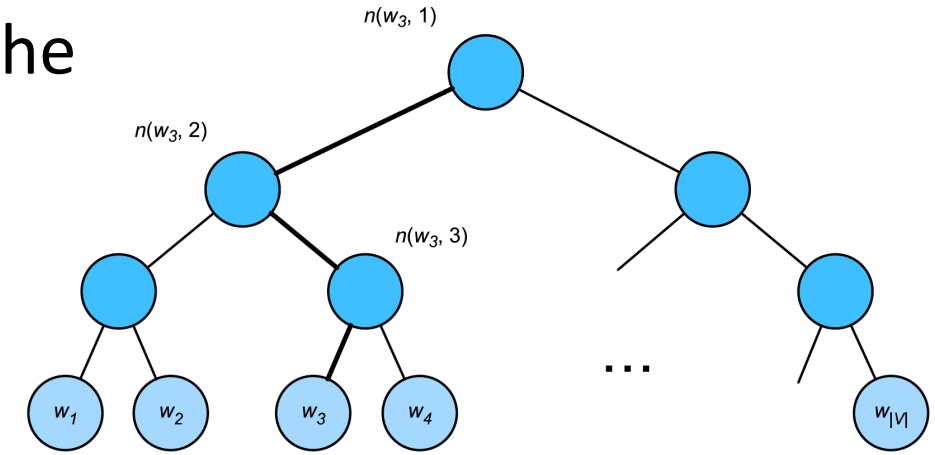
Hierarchical Softmax

- Hierarchical softmax is another type of approximate training method.
- It uses a binary tree for data structure, with the leaf nodes of the tree representing every word in the dictionary



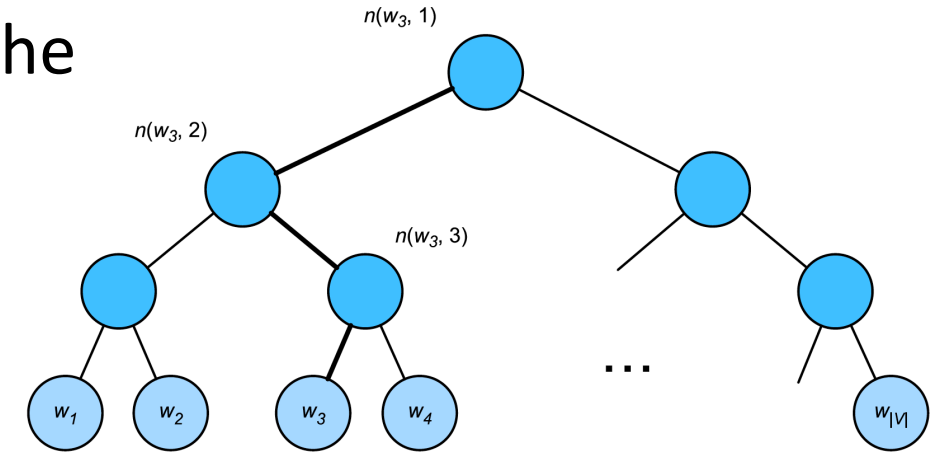
Hierarchical Softmax

- We assume that $L(w)$ is the number of nodes on the path from the root node of the binary tree to the leaf node of word w, e.g. $L(w_3) = 4$ in the figure.



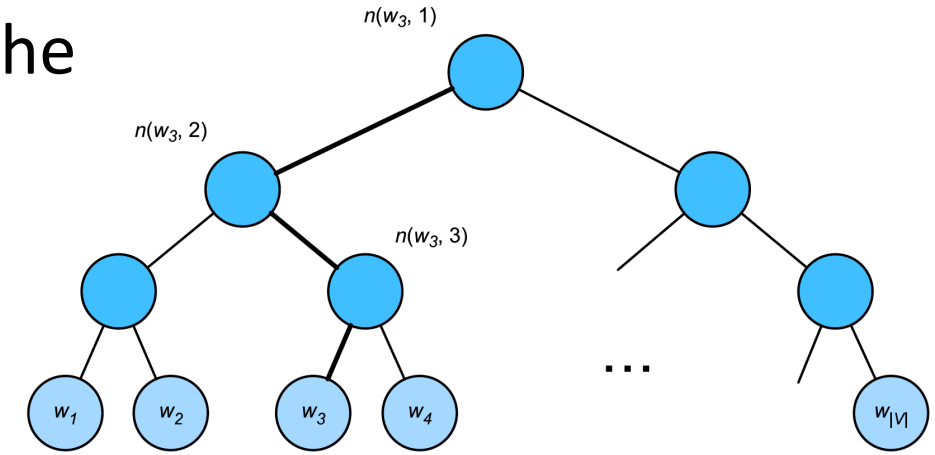
Hierarchical Softmax

- We assume that $L(w)$ is the number of nodes on the path from the root node of the binary tree to the leaf node of word w, e.g. $L(w_3) = 4$ in the figure.
- Let $n(w, j)$ be the jth node on this path, with the context word vector $\mathbf{u}_{n(w,j)}$.



Hierarchical Softmax

- We assume that $L(w)$ is the number of nodes on the path from the root node of the binary tree to the leaf node of word w, e.g. $L(w_3) = 4$ in the figure.
- Let $n(w, j)$ be the jth node on this path, with the context word vector $\mathbf{u}_{n(w,j)}$.

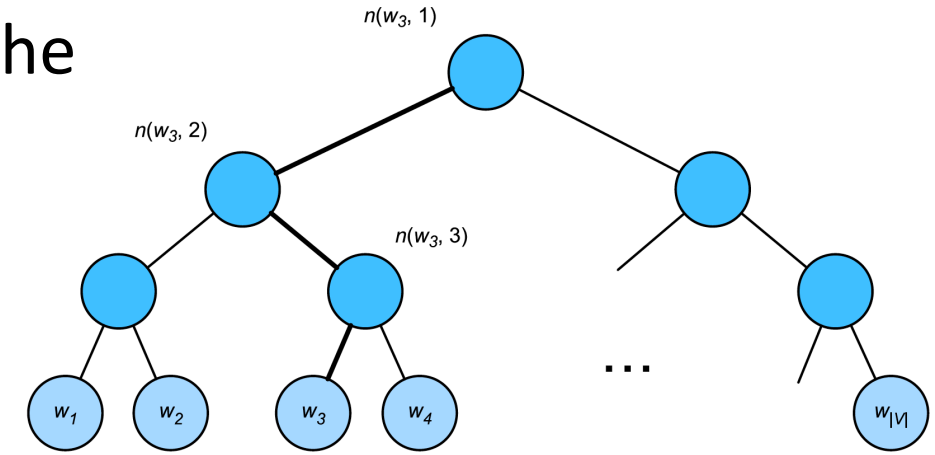


- Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$\mathbb{P}(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left(\left[\!\left[n(w_o, j+1) = \text{leftChild}(n(w_o, j))\right]\!\right] \cdot \mathbf{u}_{n(w_o,j)}^{\top} \mathbf{v}_c\right)$$

Hierarchical Softmax

- We assume that $L(w)$ is the number of nodes on the path from the root node of the binary tree to the leaf node of word w, e.g. $L(w_3) = 4$ in the figure.
- Let $n(w, j)$ be the jth node on this path, with the context word vector $\mathbf{u}_{n(w,j)}$.



- Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$\mathbb{P}(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left(\left[\!\left[n(w_o, j+1) = \text{leftChild}(n(w_o, j))\right]\!\right] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c\right)$$

- Now, we can compute the conditional probability:

$$\mathbb{P}(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c).$$

Word Embedding with Global Vectors (GloVe)

- First, we should review the skip-gram model in word2vec.
- The conditional probability $P(w_j|w_i)$ expressed in the skip-gram model using the softmax operation will be recorded as q_{ij}, that is:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)},$$

- where v_i and u_i are the vector representations of word w_i of index i as the center word and the context word, respectively.
- For word w_i, it may appear in the data set for multiple times.
- We collect all the context words every time when w_i is a center word and keep duplicates, denoted as multiset C_i
- The number of an element in a multiset is called the multiplicity of the element.

Word Embedding with Global Vectors (GloVe)

- For instance, suppose that word w_i appears twice in the data set: the context windows when these two w_i become center words in the text sequence contain context word indices $[2, 1, 5, 2]$ and $[2, 3, 2, 1]$.
- Then, multiset $C_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$ where multiplicity of element 1 is 2, multiplicity of element 2 is 4, and multiplicities of elements 3 and 5 are both 1.
- Denote multiplicity of element j in multiset C_i as x_{ij}: it is the number of word w_j in all context windows for center word w_i in the entire dataset.
- As a result, the loss function of the skip-gram model can be expressed in a different way: $-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} x_{ij}\log q_{ij}$.
- We add up the number of all the context words for the central target word w_i to get x_i, and record the conditional probability x_{ij}/x_i as p_{ij}.
- We can rewrite the loss function of the skip-gram model as $-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij}\log q_{ij}$.

Word Embedding with Global Vectors (GloVe)

- In this formula, $-\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ is the cross-entropy between the conditional probability distribution p_{ij} for context word generation based on the central target word w_i and the cross-entropy of conditional probability distribution q_{ij} predicted by model.
- The loss function is weighted using the sum of the number of context words with the central target word w_i.
- However, the cross-entropy loss function is sometimes not a good choice.
  - On the one hand the cost of letting the model prediction q_{ij} has the sum of all items in the entire dictionary in its denominator. This can easily lead to excessive computational overhead.
  - On the other hand, there are often a lot of uncommon words in the dictionary, and they appear rarely in the data set. In the cross-entropy loss function, the final prediction of the conditional probability distribution on a large number of uncommon words is likely to be inaccurate.

The GloVe Model

- To address this, GloVe, a word embedding model that came after word2vec, adopts square loss and makes three changes to the skip-gram model based on this loss.
  1. Here, we use the non-probability distribution variables $p'_{ij} = x_{ij}$ and $q'_{ij} = \exp(u_j^T v_i)$ and take their logs. Therefore, we get the square loss $\left(\log p'_{ij} - \log q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij}\right)^2$
  2. We add two scalar model parameters for each word w_i: the bias terms b_i (for central target words) and c_i (for context words).
  3. Replace the weight of each loss with the function $h(x_{ij})$. The weight function $h(x)$ is a monotone increasing function with the range $[0, 1]$
- Therefore, the goal of GloVe is to minimize the loss function.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij}\right)^2$$

The GloVe Model

- Here, we have a suggestion for the choice of weight function $h(x)$: when $x < c$ (e.g. $c = 100$) make $h(x) = \left(\frac{x}{c}\right)^{\alpha}$ (e.g. $\alpha = 0.75$), otherwise make $h(x) = 1$.
- Because $h(0) = 0$, the squared loss term for $x_{ij} = 0$ can simply be ignored.
- The non-zero x_{ij} are computed in advance based on the entire dataset and they contain global statistics for the data set.
- Therefore, the name GloVe is taken from “Global Vectors”.
- Notice that if word w_i appears in the context window of word w_j, then w_j will also appear in the context window of w_i. Therefore, $x_{ij} = x_{ji}$.
- Unlike word2vec, GloVe fits the symmetric $\log x_{ij}$ while the conditional probabilities p_{ij} are asymmetric.
- Therefore, the central target word vector and context word vector of any word are equivalent in GloVe.
- However, the two sets of word vectors that are learned by the same word may be different in the end due to different initialization values.
- After learning all the word vectors, GloVe will use the sum of the central target word vector and the context word vector as the final word vector for the word.

Understanding GloVe from Conditional Probability Ratios

- From a real example from a large corpus, here we have the following two sets of conditional probabilities with “ice” and “steam” as the central target words and the ratio between them:

$w_k =$	“solid”	“gas”	“water”	“fashion”
$p_1 = \mathbb{P}(w_k \mid \text{“ice”})$	0.00019	0.000066	0.003	0.000017
$p_2 = \mathbb{P}(w_k \mid \text{“steam”})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

- We can see that the conditional probability ratio can represent the relationship between different words more intuitively.
- We can construct a word vector function to fit the conditional probability ratio more effectively.
- The conditional probability ratio with w_i as the central target word is p_{ij}/p_{ik}.
- We can find a function that uses word vectors to fit this conditional probability ratio. $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}$.

Understanding GloVe from Conditional Probability Ratios

- The possible design of function f here will not be unique.
- We only need to consider a more reasonable possibility.
- Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$
- One possibility is to use exponential function so that the ratio is 1 when $j = k$.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}.$$

- One possibility that satisfies the right side of the approximation is $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$

where α is a constant.

- Considering that $p_{ij} = x_{ij}/x_i$, after taking the logarithm we get

$$\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$$

Understanding GloVe from Conditional Probability Ratios

$$\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$$

- We use additional bias terms to fit $\log \alpha - \log x_i$, such as the central target word bias term b_i and context word bias term c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij})$$

- By using formula below, we can get the loss function of GloVe.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij} \right)^2$$

Text Classification

- Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text.

Text Classification

- Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text.
- By using text sentiment classification we can analyze the emotions of the text’s author.

Text Classification

- Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text.
- By using text sentiment classification we can analyze the emotions of the text’s author.
- This problem is also called sentiment analysis and has a wide range of applications.

Text Classification

- Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text.
- By using text sentiment classification we can analyze the emotions of the text's author.
- This problem is also called sentiment analysis and has a wide range of applications.
- For example, we can analyze user reviews of products to obtain user satisfaction statistics, or analyze user sentiments about market conditions and use it to predict future trends.

Text Sentiment Classification Data

- We use Stanford's Large Movie Review Dataset as the data set for text sentiment classification
- This dataset is divided into two data sets for training and testing purposes, each containing 25,000 movie reviews downloaded from IMDb.
- In each data set, the number of comments labeled as "positive" and "negative" is equal.

Reading Data

- Data is downloaded from Stanford's server

```
# Save to the d2l package.
def download_imdb(data_dir='../data'):
    url = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
    fname = gluon.utils.download(url, data_dir)
    with tarfile.open(fname, 'r') as f:
        f.extractall(data_dir)

download_imdb()
```

Reading Data

- Next, read the training and test data sets. Each example is a review and its corresponding label: 1 indicates “positive” and 0 indicates “negative”.

```
# Save to the d2l package.
def read_imdb(folder='train', data_dir='../data'):
    data, labels = [], []
    for label in ['pos', 'neg']:
        folder_name = os.path.join(data_dir, 'aclImdb', folder, label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb('train')
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

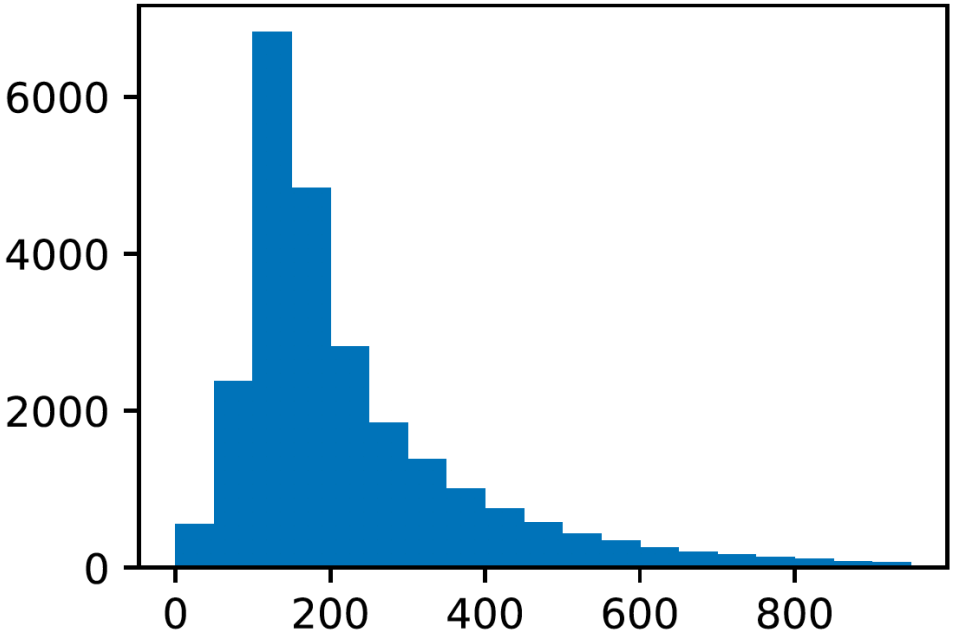
```
# trainings: 25000
label: 1 review: Normally the best way to annoy
label: 1 review: The Bible teaches us that the
label: 1 review: Being someone who lists Night
```

Tokenization and Vocabulary

- We use a word as a token, and then create a dictionary based on the training data set.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5)

d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0,1000,50));
```



Padding to the Same Length

- Because the reviews have different lengths, so they cannot be directly combined into mini-batches. Here we fix the length of each comment to 500 by truncating or adding “<unk>” indices.

```
num_steps = 500  # sequence length
train_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                           for line in train_tokens])
train_features.shape
```

```
(25000, 500)
```

Create Data Iterator

- Now, we will create a data iterator. Each iteration will return a mini-batch of data.

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'# batches:', len(train_iter)
```

```
X (64, 500) y (64,)
```

```
('# batches:', 391)
```

Put All Things Together

```
# Save to the d2l package.
def load_data_imdb(batch_size, num_steps=500):
    download_imdb()
    train_data, test_data = read_imdb('train'), read_imdb('test')
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                               for line in train_tokens])
    test_features = nd.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                              for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

Text Sentiment Classification: Using Recurrent Neural Networks

- Text classification is also a downstream application of word embedding.
- Here, we will apply pre-trained word vectors and bidirectional recurrent neural networks with multiple hidden layers.
- We will use them to determine whether a text sequence of indefinite length contains positive or negative emotion.

```
import d2l
from mxnet import gluon, init, nd
from mxnet.gluon import nn, rnn
from mxnet.contrib import text

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

Use a Recurrent Neural Network Model

- Each word first obtains a feature vector from the embedding layer

Use a Recurrent Neural Network Model

- Each word first obtains a feature vector from the embedding layer
- Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information.

Use a Recurrent Neural Network Model

- Each word first obtains a feature vector from the embedding layer
- Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information.
- Finally, we transform the encoded sequence information to output through the fully connected layer.

Use a Recurrent Neural Network Model

- Each word first obtains a feature vector from the embedding layer
- Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information.
- Finally, we transform the encoded sequence information to output through the fully connected layer.

```
class BiRNN(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set Bidirectional to True to get a bidirectional recurrent neural
        # network
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)
```

Load Pre-trained Word Vectors

- We will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words.

Load Pre-trained Word Vectors

- We will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words.
- Here, we load a 100-dimensional GloVe word vector for each word in the dictionary vocab.

```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
```

Query the word vectors that in our vocabulary.

```
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
embeds.shape
```

```
(49339, 100)
```

Load Pre-trained Word Vectors

- We will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words.
- Here, we load a 100-dimensional GloVe word vector for each word in the dictionary vocab.

```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
```

Query the word vectors that in our vocabulary.

```
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
embeds.shape
```

```
(49339, 100)
```

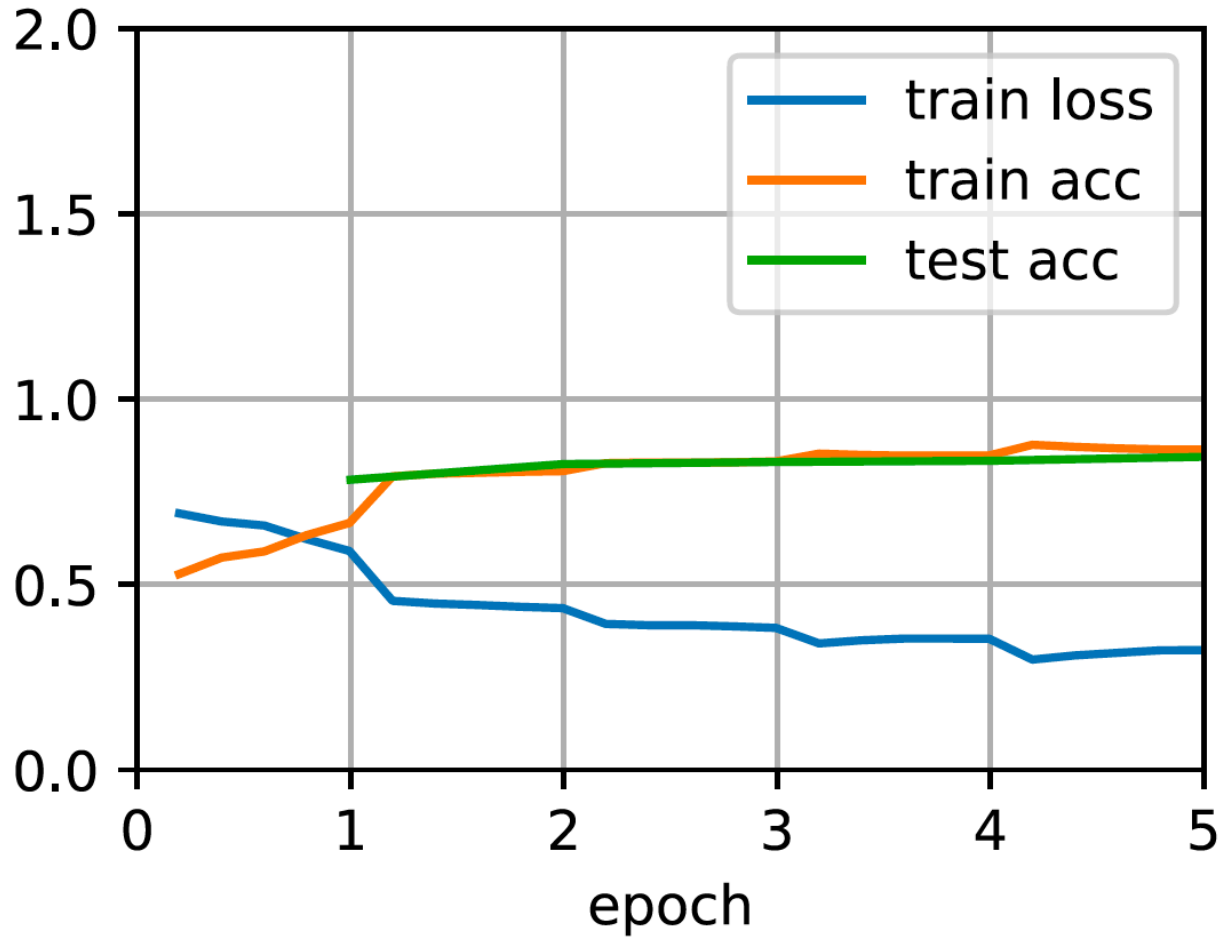
- Then, we will use these word vectors as feature vectors for each word in the reviews.

```
net.embedding.weight.set_data(embeds)
net.embedding.collect_params().setattr('grad_req', 'null')
```

Training

```
lr, num_epochs = 0.01, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch12(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.323, train acc 0.864, test acc 0.845
887.7 exampes/sec on [gpu(0), gpu(1)]
```



Prediction

```
# Save to the d2l package.
def predict_sentiment(net, vocab, sentence):
    sentence = nd.array(vocab[sentence.split()], ctx=d2l.try_gpu())
    label = nd.argmax(net(sentence.reshape((1, -1))), axis=1)
    return 'positive' if label.asscalar() == 1 else 'negative'
```

Then, use the trained model to classify the sentiments of two simple sentences.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

- We explored how to process two-dimensional image data with two-dimensional convolutional neural networks.

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

- We explored how to process two-dimensional image data with two-dimensional convolutional neural networks.
- In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data.

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

- We explored how to process two-dimensional image data with two-dimensional convolutional neural networks.
- In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data.
- In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words.

Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

- We explored how to process two-dimensional image data with two-dimensional convolutional neural networks.
- In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data.
- In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words.
- Here we describe a groundbreaking approach to applying convolutional neural networks to text analysis: textCNN

One-dimensional Convolutional Layer

- Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional crosscorrelation operation.

One-dimensional Convolutional Layer

- Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional crosscorrelation operation.
- In the one-dimensional cross-correlation operation, the convolution window starts from the leftmost side of the input array and slides on the input array from left to right successively.

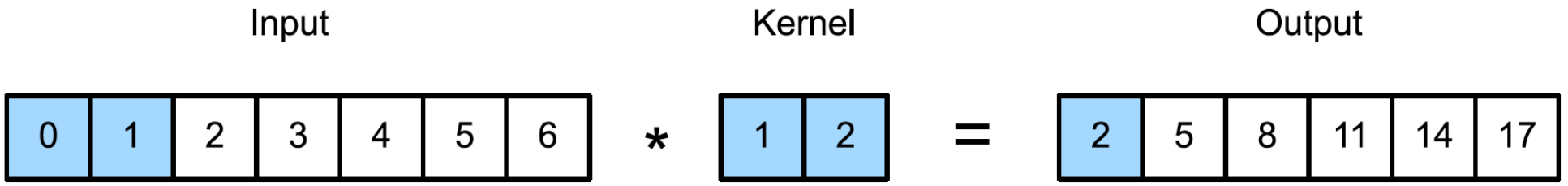


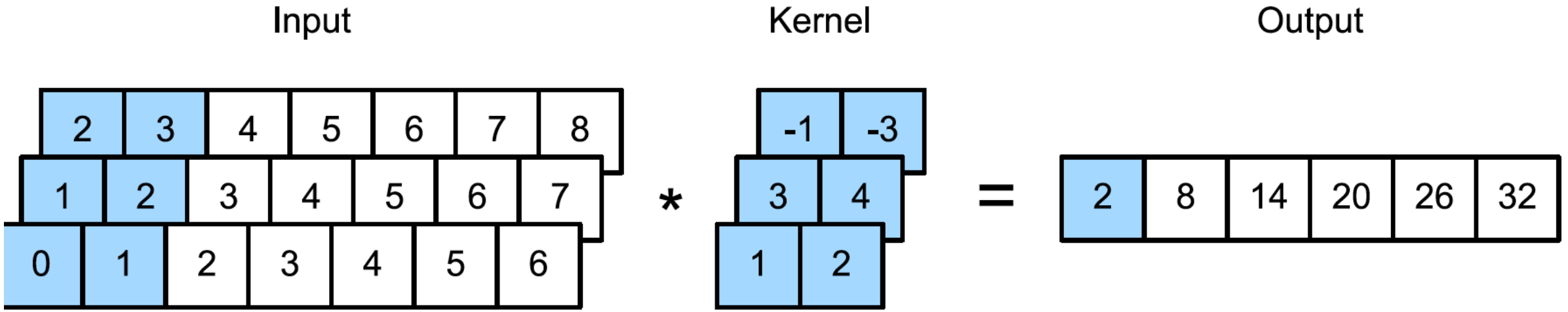
Fig. 15.10.1: One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 = 2$.

One-dimensional Convolutional Layer

- The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels.

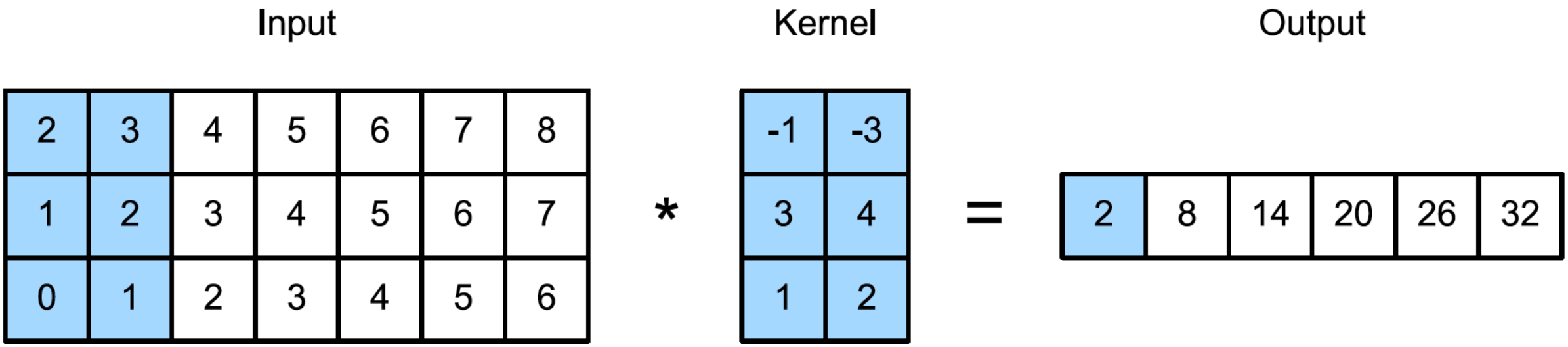
One-dimensional Convolutional Layer

- The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels.
- On each channel, it performs the one-dimensional cross-correlation operation on the kernel and its corresponding input and adds the results of the channels to get the output.



One-dimensional Convolutional Layer

- The definition of a two-dimensional cross-correlation operation tells us that a one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.



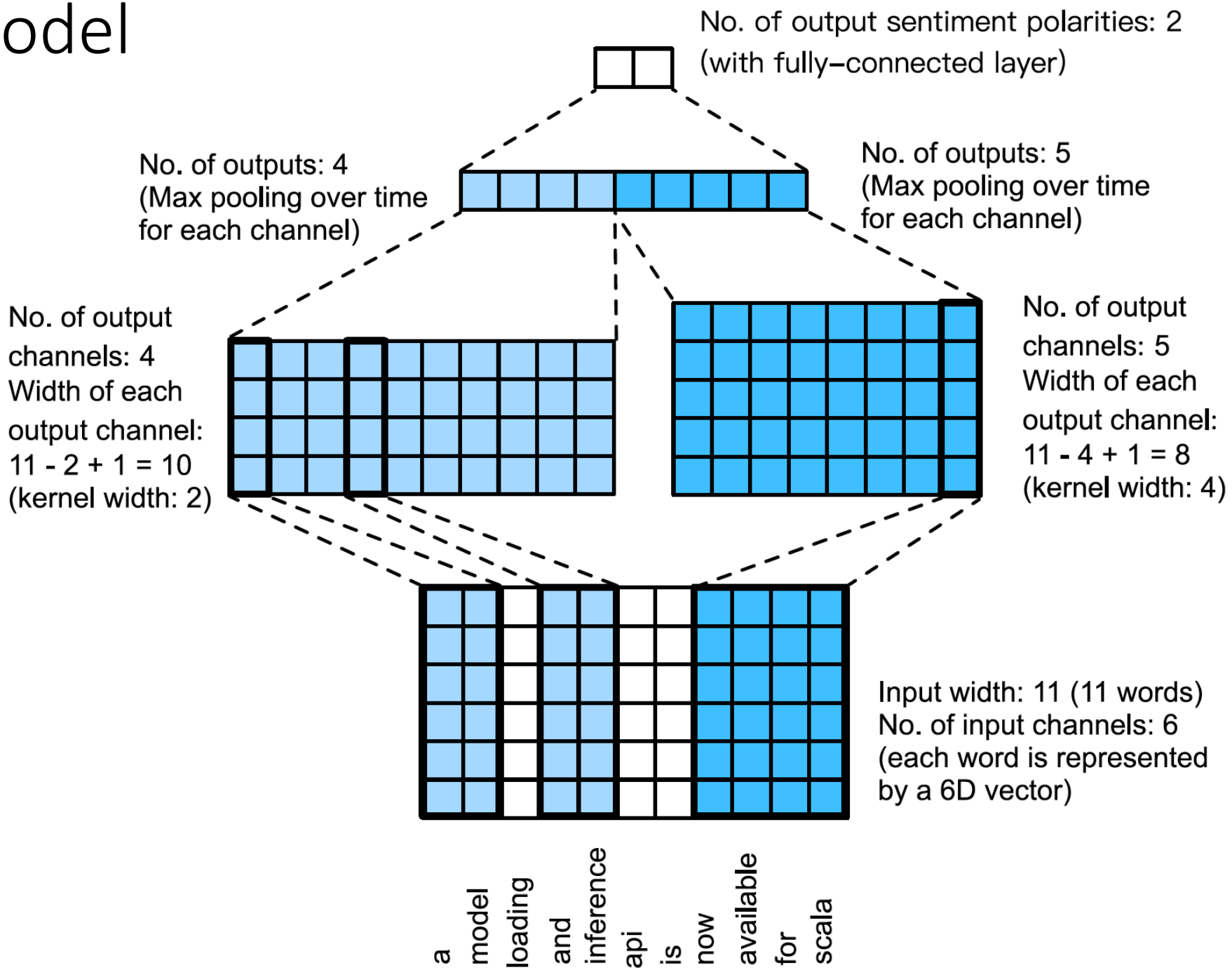
Max-Over-Time Pooling Layer

- Similarly, we have a one-dimensional pooling layer.
- The max-over-time pooling layer used in TextCNN actually corresponds to a one-dimensional global maximum pooling layer.
- Assuming that the input contains multiple channels, and each channel consists of values on different time steps, the output of each channel will be the largest value of all time steps in the channel

The TextCNN Model

- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.
- Suppose the input text sequence consists of n words, and each word is represented by a d-dimension word vector.
- Then the input example has a width of n, a height of 1, and d input channels.
- The calculation of textCNN can be mainly divided into the following steps:
  1. Define multiple one-dimensional convolution kernels and use them to perform convolution calculations on the inputs.
  2. Perform max-over-time pooling on all output channels, and then concatenate the pooling output values of these channels in a vector.
  3. The concatenated vector is transformed into the output for each category through the fully connected layer. A dropout layer can be used in this step to deal with overfitting.

The TextCNN Model



The End of EE-628- Fall 2019

Thank you for your attention and patience

Any Questions?