Trained using unpaired data. Why unpaired?

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

Of course, labels for a horse to zebra transformation do not exist!

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks

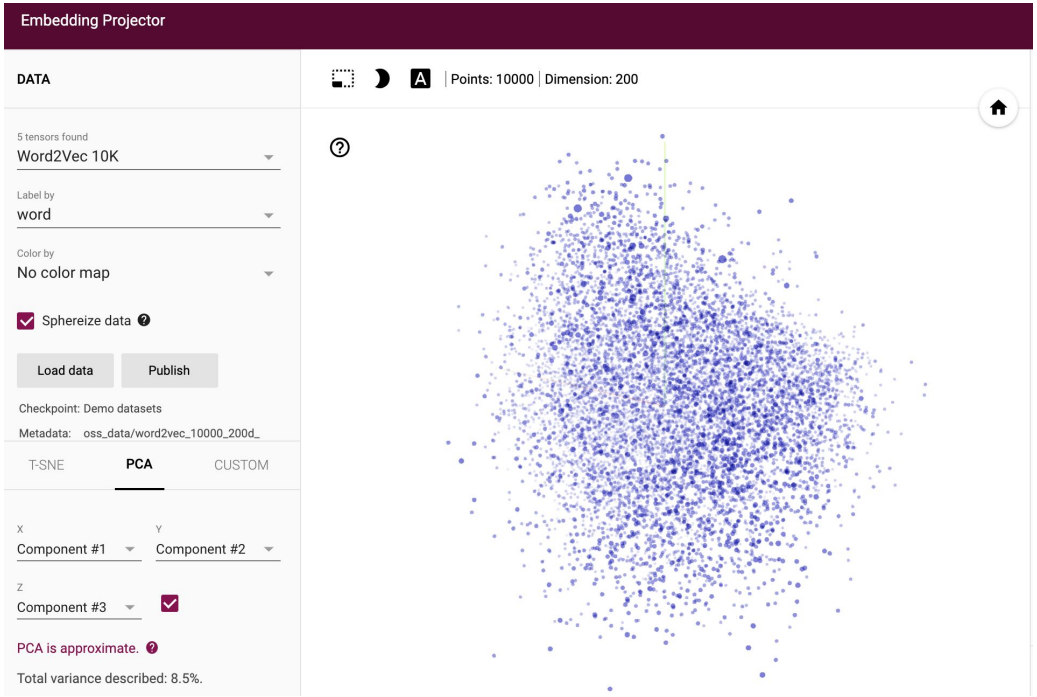
Generative models

Applied Deep Learning • April 25th, 2019

Agenda

- HW5 EC idea (sentence embeddings with the projector
- Misc new examples
- GANs, VAEs
- Transposed Convolution
- Batch Norm
- Data and model parallelism
- Karpathy's article

HW5 EC idea: sentence embeddings



http://projector.tensorflow.org/

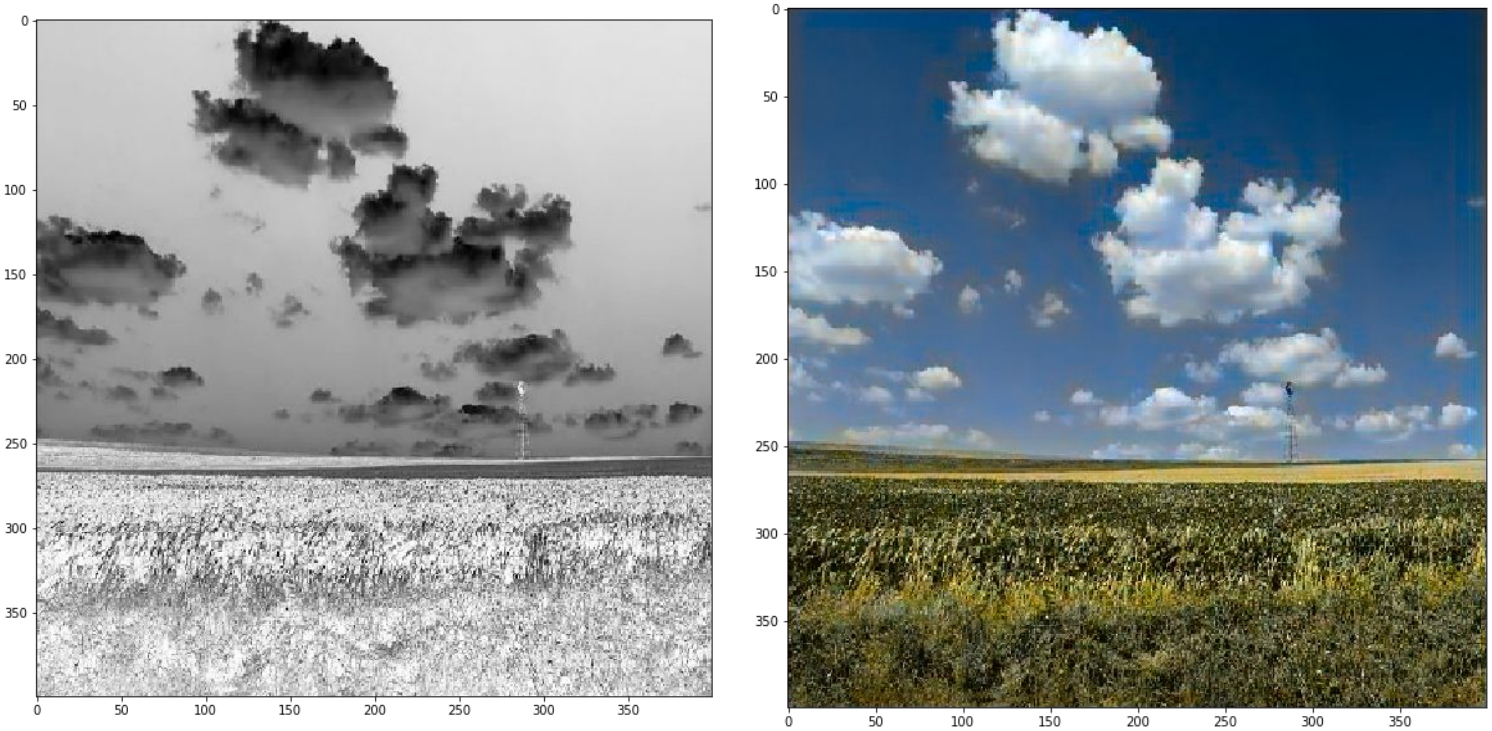
Administrative stuff

Project presentations are **optional** next week

- Please bring questions if you're not finished, happy to help point you in the right direction.

Reminder: projects due on 5/19 including a YouTube video presentation (can be unlisted) and code.

Image colorization (minimal example)



github.com/random-forests/applied-dl/blob/master/examples/9-image-colorization.ipynb

Style Transfer (cleaned up example)

A Neural Algorithm of Artistic Style
https://www.tensorflow.org/alpha/tutorials/generative/style_transfer

Explain style transfer

Quickly walk through the basic idea

- You're probably tired of me saying that "lower layers in CNNs detect features like shapes, higher layers detect increasing abstract features" …
- … but that's exactly what this technique exploits.

Note: the best current way to do Style Transfer is using a GAN (why develop a custom loss function, when you can learn one algorithmically?)

Themes

- Representation learning
- Deep learning as compression

We spoke about these before with image captioning. GANs and VAEs offer another view.

- Compress an image to a sentence (seq2seq)
- Use a short, random vector to seed image generation (DCGAN).

Generative Adversarial Networks

GANs

Classifying images is easy

- Why? We have a **loss function** we can optimize against (cross entropy).

Generating images is hard

- What loss will tell us if a **generated image** is real or fake?
- How could you write this? What could you do if you had it?

Goal is to output 1 for images from the training set, and 0 for fakes.

Discriminator network

Just a run-of-the-mill CNN. (Other designs are possible, too).

Real images (from the training set)

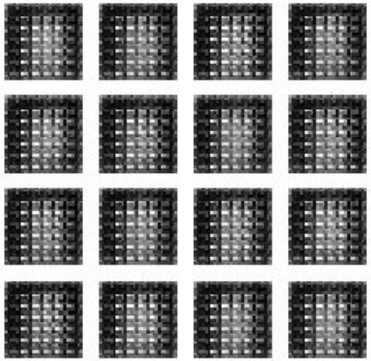
Fake images (produced by the generator)

Generator network

Turns a vector from the latent space into an image (by upsampling, more on that in a sec).

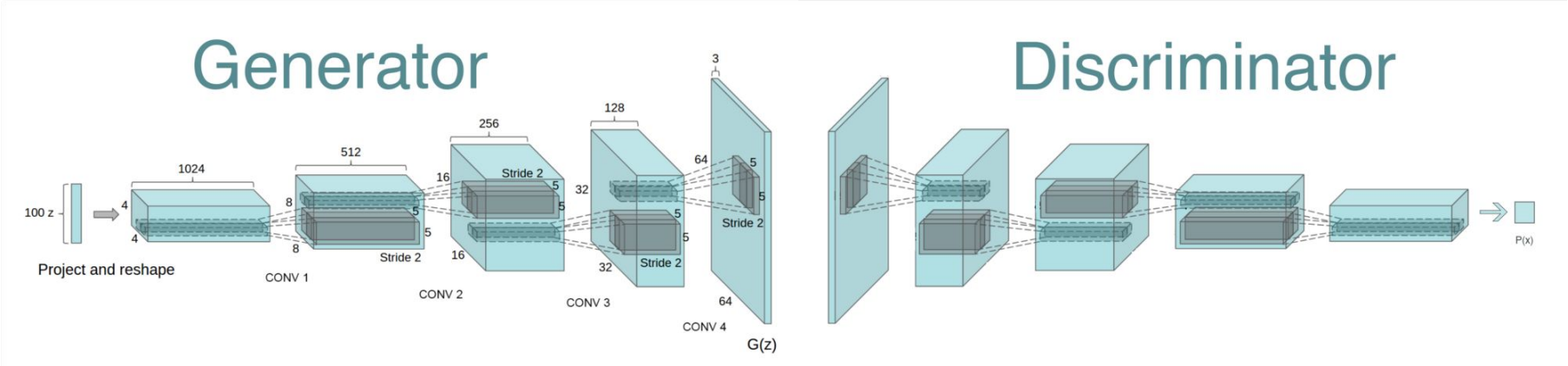
Input noise z

Generative Adversarial Networks (2014)



Images generated as this notebook is trained for 50 epochs.

tensorflow.org/alpha/tutorials/generative/dcgan



Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (2015)

Learned filters from the last convolutional layer of the discriminator

Discriminator (probably looks familiar)

```
class Discriminator(tf.keras.Model):
  def __init__(self):
    super(Discriminator, self).__init__()
    self.conv1 = Conv2D(64, (5, 5), strides=(2, 2), padding='same')
    self.dropout = Dropout(0.3)
    self.flatten = Flatten()
    self.fc1 = Dense(1)

  def call(self, x, training=True):
    x = tf.nn.leaky_relu(self.conv1(x))
    x = self.dropout(x, training=training)
    x = self.flatten(x)
    x = self.fc1(x)
    return x
```

No activation on final layer (that's handled by the loss function we happen to use).

You may want to use a more powerful discriminator in practice, but this will get us going.

Discriminator Loss

```
def discriminator_loss(real_output, fake_output):

      We want predictions for real images to be close to 1...

    real_loss = cross_entropy(tf.ones_like(real_output), real_output)

      ... and predictions for fake images to be close to 0.

    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)

    total_loss = real_loss + fake_loss

    return total_loss
```

Generator (just a couple things are new)

```
class Generator(tf.keras.Model):
  def __init__(self):
    super(Generator, self).__init__()
    self.fc1 = Dense(7*7*64, use_bias=False)
    self.batchnorm1 = BatchNormalization()
    self.conv1 = Conv2DTranspose(64, (5, 5), strides=(1, 1), padding='same')
    self.batchnorm2 = BatchNormalization()
    self.conv2 = Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same')
    self.batchnorm3 = BatchNormalization()
    self.conv3 = Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same')
```

Recall convolution

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	

Output image
(after convolving with stride 1)

Recall convolution

2	0	1	1
0	1	0	0
0	0	1	0
0	3	0	0

An input image
(no padding)

1	0	1
0	0	0
0	1	0

A filter
(3x3)

3	2

Output image
(after convolving with stride 1)

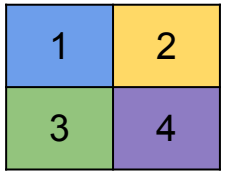
Transposed Convolution (upsampling layers)

Transposed convolution

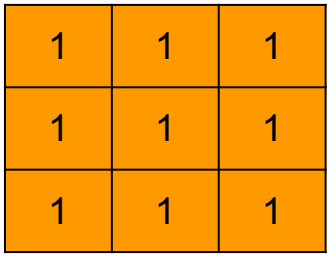
Also referred to as "deconvolution" or "fractionally strided convolution".

*Stride 2 with a 3x3 filter and padding 'same'. Output size = stride * input_size. At each step, the value of the input image is used to weight the filter. The result is copied to the output image.*

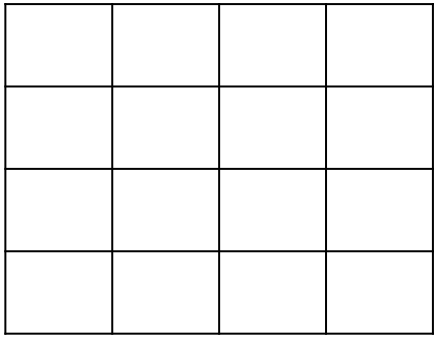
The value of each cell below is the image pixel (on the left) multiplied by the filter value. Sum where overlaps.



Input image (2x2)



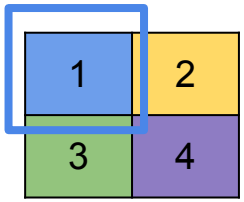
Filter (3x3), learned weights - initialized to one's for our example



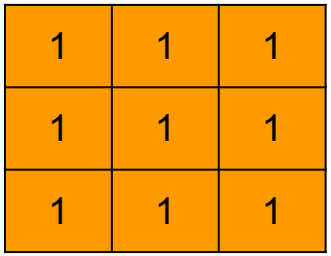
Output image (4x4)

A guide to convolution arithmetic for deep learning — *Probably more than you ever wanted to read about convolution.*

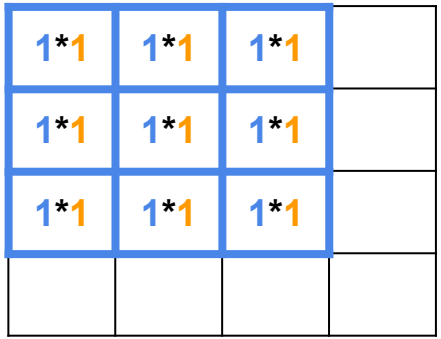
```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



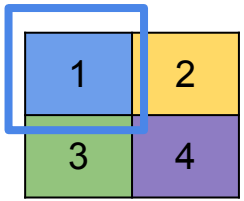
Filter (3x3), learned weights - initialized to one's for our example



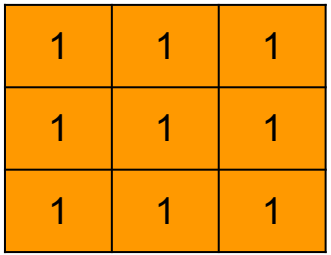
Output image (4x4)

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

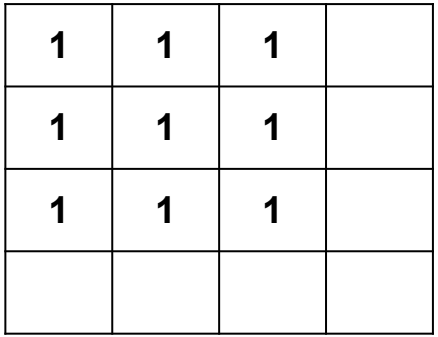
```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



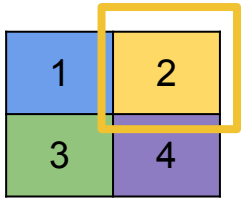
Filter (3x3), learned weights - initialized to one's for our example



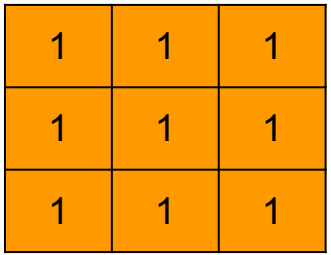
Output image (4x4)

A guide to convolution arithmetic for deep learning ← Probably more than you ever wanted to read about convolution.

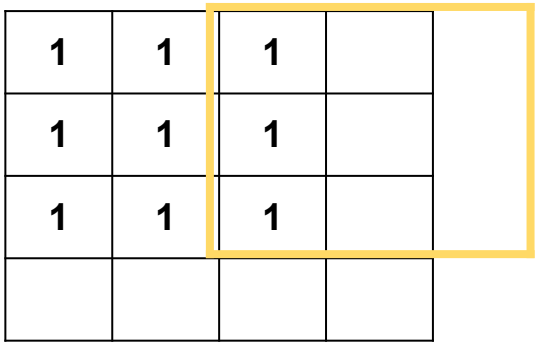
```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



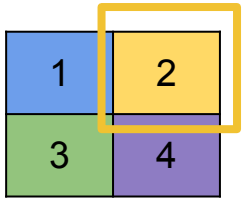
Filter (3x3), learned weights - initialized to one's for our example



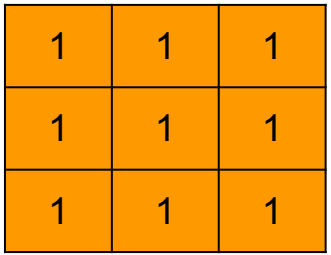
Output image (4x4)

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```

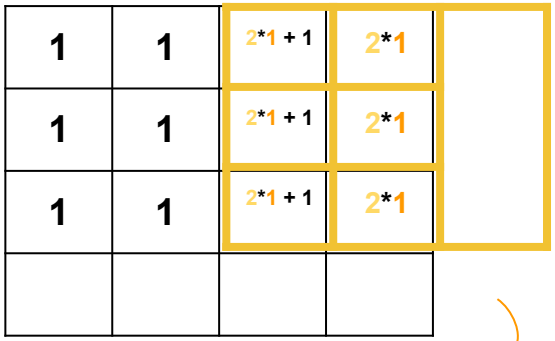


Input image (2x2)



Filter (3x3), learned weights - initialized to one's for our example

Sum at overlap

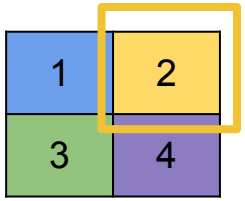


Output image (4x4)

This region will be trimmed.

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)

1	1	1
1	1	1
1	1	1

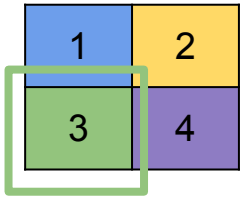
Filter (3x3), learned weights - initialized to one's for our example

1	1	3	2
1	1	3	2
1	1	3	2

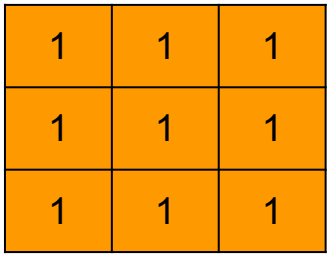
Output image (4x4)

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

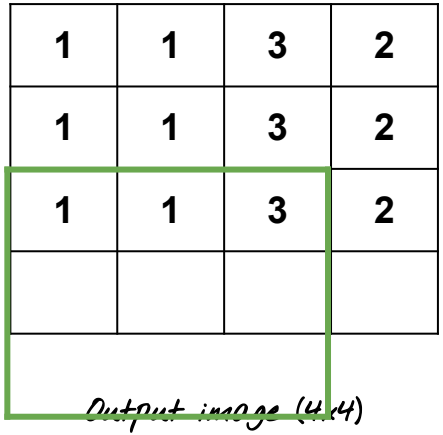
```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



Filter (3x3), learned weights - initialized to one's for our example

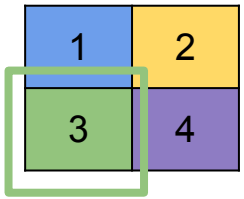


Output image (4x4)

This region will be trimmed.

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)

1	1	1
1	1	1
1	1	1

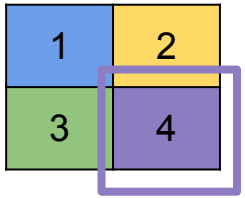
Filter (3x3), learned weights - initialized to one's for our example

1	1	3	2
1	1	3	2
4	4	6	2
3	3	3	

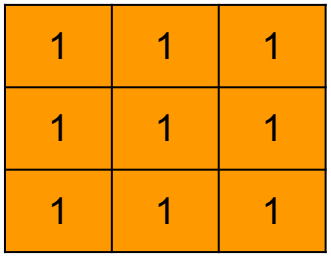
Output image (4x4)

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

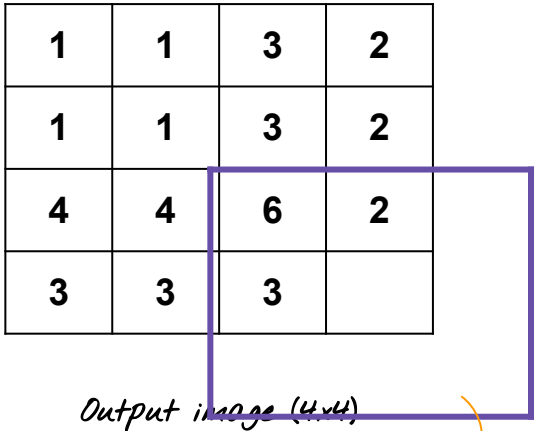
```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



Filter (3x3), learned weights - initialized to one's for our example

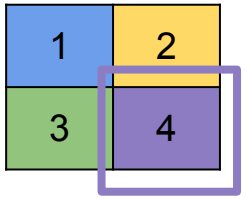


Output image (4x4)

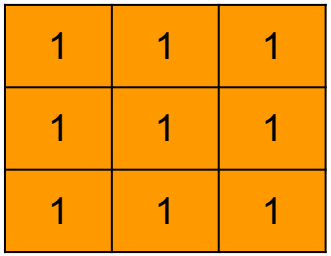
This region will be trimmed.

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

```
layer = Conv2DTranspose(filters=1,kernel_size=3,
                        strides=(2,2), padding='same',
                        kernel_initializer=ones)
```



Input image (2x2)



Filter (3x3), learned weights - initialized to one's for our example

1	1	3	2
1	1	3	2
4	4	10	6
3	3	7	4

Output image (4x4)

A guide to convolution arithmetic for deep learning — Probably more than you ever wanted to read about convolution.

Batch normalization

Where can I find practical examples?

- tensorflow.org/alpha/tutorials/generative/dcgan

Batch Normalization example

```
# Usually used right before the activation
def call(self, x, training=True):

    # with a dense layer, like this:
    x = self.dense1(x) # dense
    x = self.batchnorm1(x, training=training) # bn
    x = tf.nn.relu(x) # activation

    # ... or with a conv layer, like this:
    x = self.conv1(x) # conv
    x = self.batchnorm2(x, training=training) # bn
    x = tf.nn.relu(x) # activation
    return x
```

Two things to to notice

- *We pass a training flag to the call method (like Dropout, the behavior of BN is different during training and inference)*
- *We create a separate BN layer for each layer we apply it to (BN layers compute statistics about the layer directly below them)*

[Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift]

Intuition

Consider this network.

$$\texttt{loss} = F_2(F1_1(x, \Theta_1), \Theta_2)$$

1. **Updating Θ_1 …**
2. **Changes the input distribution to F_2**

As we train a network with SGD, we have to keep in mind that:

1. Inputs to each layer are affected by the parameters of all preceding layers.
2. Adjusting the values of Θ_1 changes the distribution of inputs to F_2.
3. Subsequent layers must continuously adapt to new distributions of inputs → slower training.
4. If the distribution of outputs of F_1 remains consistent over time, then Θ_2 does not have to readjust to compensate!

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

Batch Normalization

For a layer with d-dimensional inputs

$$x = (x_i \dots x_n)$$

Normalize each by subtracting the mean and dividing by the standard deviation (computed over all values for x_i in the batch)

$$\widehat{x_i} = \frac{x_i - mean(batch(x_i))}{stdev(batch(x_i)) + \epsilon}$$

Add two learnable parameters for each input. These scale and shift the normalized value, ensuring the network could restore the original activations if that was the right thing to do.

$$BN(x_i) = \alpha\widehat{x_i} + \beta$$

At train time:

- *Each mini-batch produces estimates of the mean and variance for each feature.*

At inference time:

- *Mean / std not computed on a batch (instead, a running average is kept during training).*

Maintains a mean activation close to zero and a standard deviation close to one.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

Practical benefits

- Faster convergence.
- You can use higher learning rates.
- Network more robust to poor initialization.

Summary

- Goal: achieve a stable distribution of activation values during training (so subsequent layers don't have to continuously adjust → faster learning).
- You can think of BN as applying preprocessing at every layer of the network, with a learnable scale.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)

Back to our GAN

Goal is to output 1 for images from the training set, and 0 for fakes.

Discriminator network

Just a run-of-the-mill CNN. (Other designs are possible, too).

Real images
(from the training set)

Fake images
(produced by the generator)

Generator network

Input noise z

Turns a vector from the latent space into an image (by using upsampling convolutions, more on that in a sec).

https://papers.nips.cc/paper/5423-generative-adversarial-nets

Finishing the generator

```
def call(self, x, training=True):
  x = self.fc1(x)
  x = self.batchnorm1(x, training=training)
  x = tf.nn.relu(x)
  x = tf.reshape(x, shape=(-1, 7, 7, 64))
  x = self.conv1(x)
  x = self.batchnorm2(x, training=training)
  x = tf.nn.relu(x)
  x = self.conv2(x)
  x = self.batchnorm3(x, training=training)
  x = tf.nn.relu(x)
  return tf.nn.tanh(self.conv3(x))
```

Still 7x7

Now 14x14

Now 28x28

```
# recall
conv1 = Conv2DTranspose(... strides=(1, 1))
conv2 = Conv2DTranspose(... strides=(2, 2))
conv3 = Conv2DTranspose(... strides=(2, 2))
```

Generator loss

The Generator's loss is a function of how well it fooled the discriminator.

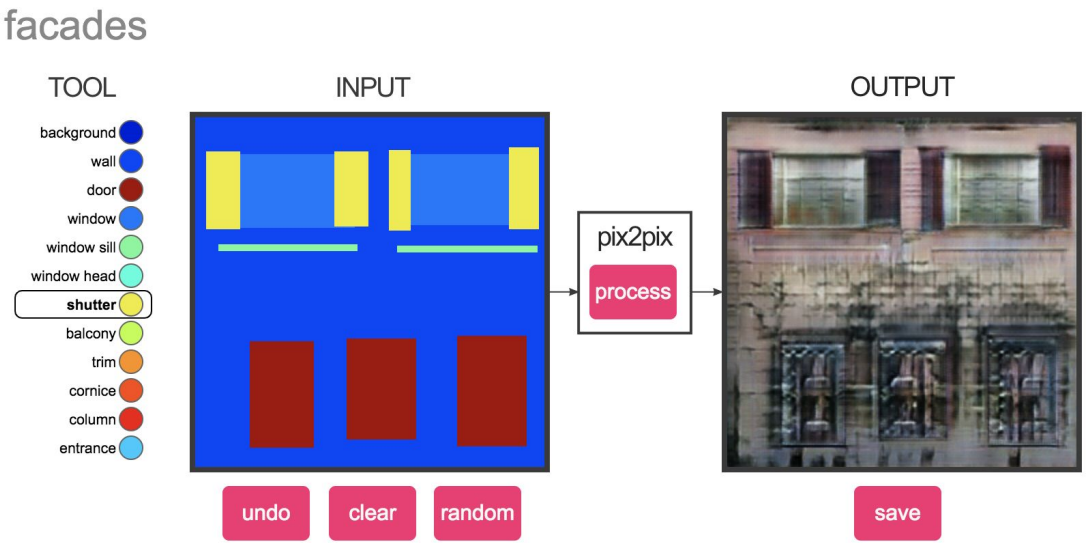
```
def generator_loss(discriminator_output):
    return tf.losses.sigmoid_cross_entropy(tf.ones_like(discriminator_output),
                                           logits=discriminator_output)
```

```
input_vector = tf.random_normal([batch_size, latent_dim])
with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:

  fake_images = generator(input_vector, training=True)
  real_guesses = discriminator(real_images, training=True)
  fake_guesses = discriminator(fake_images, training=True)

  gen_loss = generator_loss(fake_guesses)
  disc_loss = discriminator_loss(real_guesses, fake_guesses)

gen_grads = gen_tape.gradient(gen_loss, generator.variables)
disc_grads = disc_tape.gradient(disc_loss, discriminator.variables)

gen_opt.apply_gradients(zip(gen_grads, generator.variables))
disc_opt.apply_gradients(zip(disc_grads, discriminator.variables))
```

More advanced GANs

Pix2Pix - try it!

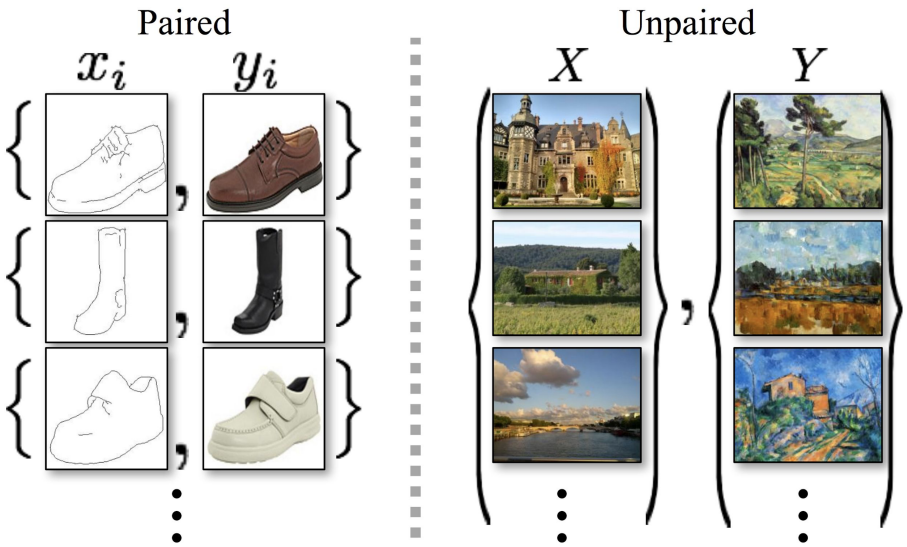
Live demo.



Complete code in TensorFlow with Keras + eager.

Image-to-Image Translation with Conditional Adversarial Networks (2016)

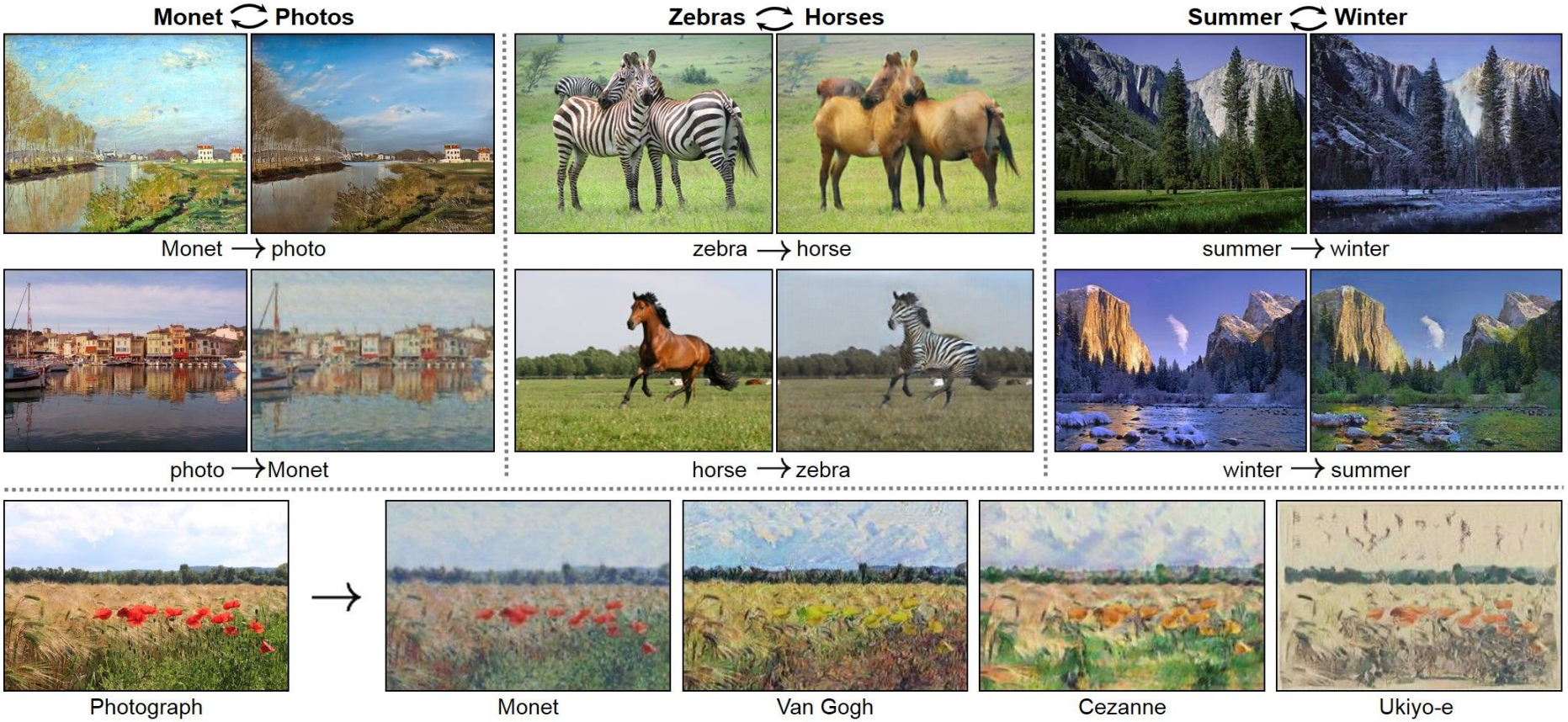
CycleGan: Learning *without* paired input-output examples



Obtaining paired training data can be difficult and expensive.

- Only a few datasets may exist for tasks we're interested in (say, semantic segmentation / Facades).
- In a dynamic environment, difficult to control for exactly the same image with different conditions (say, a busy street at day / night).

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (2017)



[Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks](#) (2017)

Algorithm

Although we lack supervision in the form of paired examples, we have supervision at the level of sets!

Given two sets of images (X, Y) in different domains (day, night):

- Train $G: X \rightarrow Y$, such that $\hat{y} = G(x)$, $x \subset X$ is indistinguishable from images y by a discriminator trained to classify $\hat{y}$ apart from y.

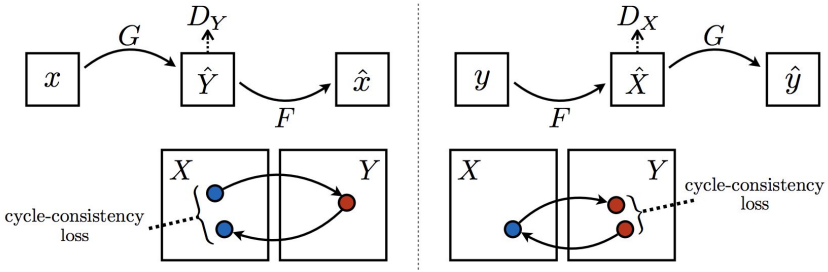
Problems:

* *Doesn't guarantee $x, \hat{y}$ are paired up in a meaningful way.*

* *Susceptible to mode collapse: all input images map to the same output image, training stalls.*

Cycle Consistency Loss

Effective use of unlabeled data is one of the most important challenges in computer science today.



Idea:

- *For each image x generated from domain X, the image translation cycle should be able to bring x back to the original image, and vice versa for y in Y.*

Adversarial loss

$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \log D_Y(y) + \log(1 - D_Y(G(x))$$

Cycle loss

$$\mathcal{L}_{\text{cyc}}(G, F) = \|F(G(x)) - x\|_1 + \|G(F(y)) - y\|_1$$

Full loss

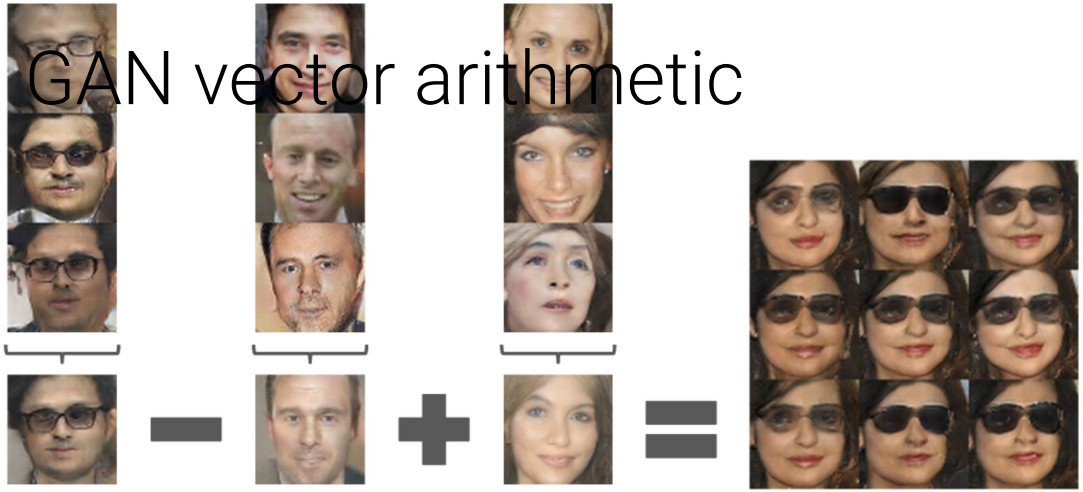
$$\begin{aligned}\mathcal{L}(G, F, D_X, D_Y) =& \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) \\ &+ \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) \\ &+ \lambda \mathcal{L}_{\text{cyc}}(G, F),\end{aligned}$$

Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (2017)

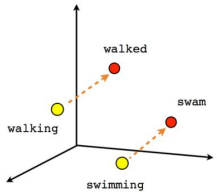
What are ‘Deep Fakes’ and what do we do about them?

- Societal problems (even if we had the tech, to identify, already damange is done)

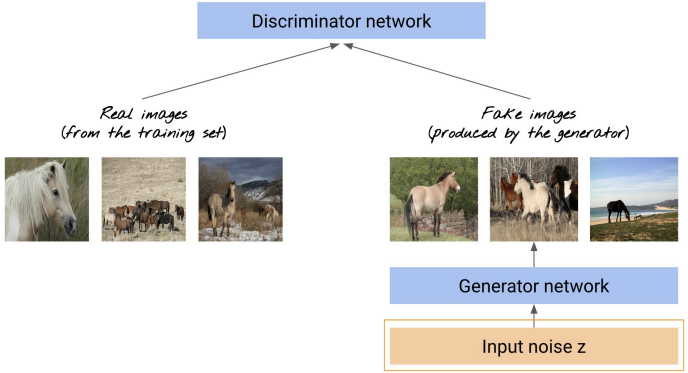
GAN vector arithmetic



Related to word embeddings, but somewhat creepier.



Verb tense



Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (2015)

Deconvolution and Checkerboard Artifacts

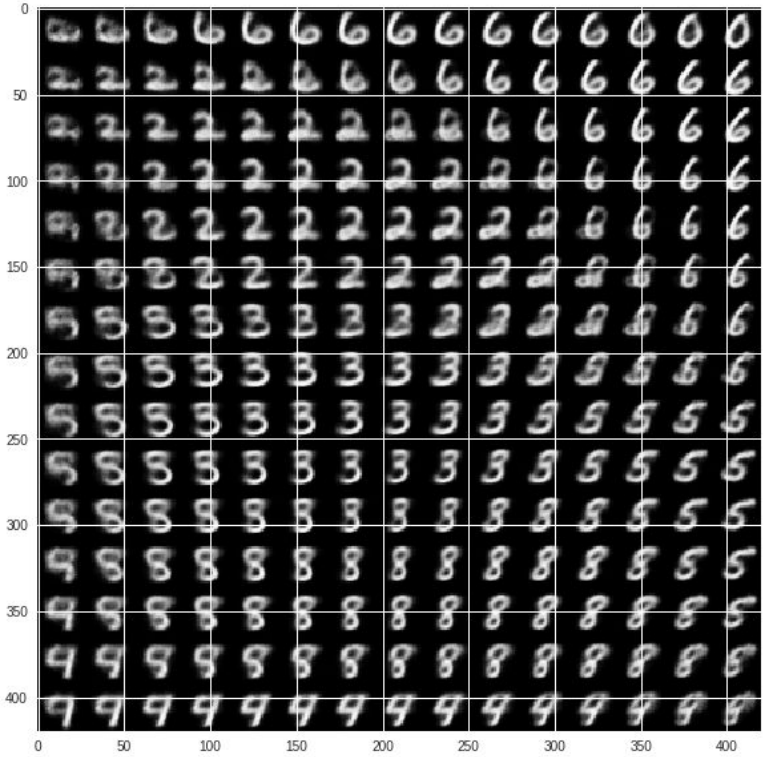
It's also possible for the generator to find "holes" in the discrimator's ability to detect fake images, resulting in patterns that look fake to us, but not the model.

Deconvolution and Checkerboard Artifacts

VAEs

Variational autoencoders

tensorflow.org/alpha/tutorials/generative/cvae



Output of our code example after training for 10 epochs.

Autoencoders

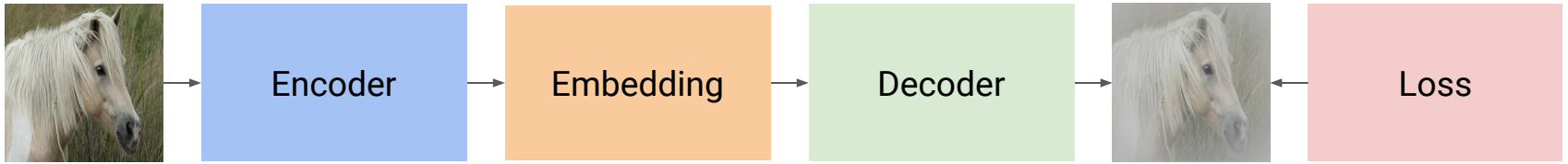
First, background - then we'll make this concrete with complete code.

Important ideas:
- Unsupervised learning
- DL as compression.

A neural network (say, a CNN). Goal: map the input image down into a lower dimensional space.

A neural network (say, a CNN). Goal: reconstruct the input image from the embedding.

Say, l2 loss on the pixel values.



Input data

A lower dimensional representation of the input data (say, a vector with 100 elements)

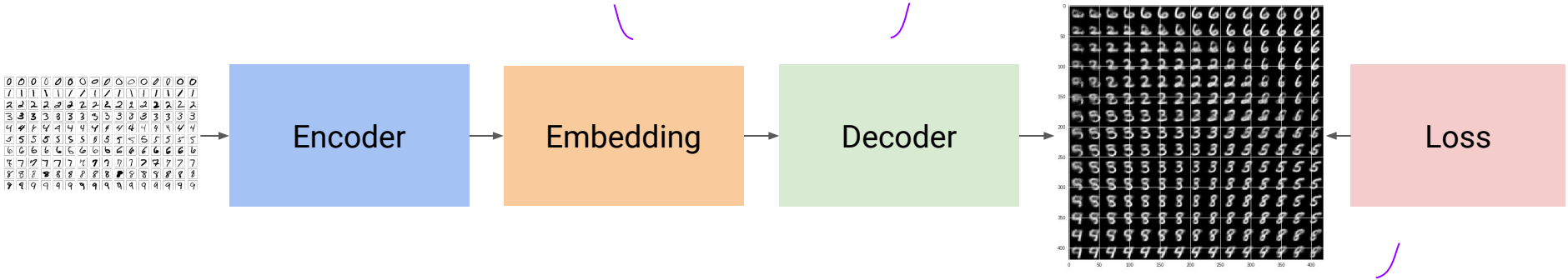
Reconstructed data (often noisy).

Variational Autoencoders

Update: can sample smoothly from the distribution to generate smoothly varying images.

Update: much smaller (say, four dimensions). Used to represent the parameters (mean and variance) of a distribution. Say we have four: could represent orientation and breed of horse.

- When sampling, a bit of noise is added —this forces every point in the embedding to have a meaningful representation.



Update: in addition to L2 loss, add a second term to keeps the elements of the embedding close to zero.

Auto-Encoding Variational Bayes (2013)

Encoder

Image space -> embedding space.

```
latent_dim = 2

class Encoder(tf.keras.Model):
  def __init__(self):
    super(Encoder, self).__init__()
    self.conv1 = Conv2D(32, 3, padding='same')
    # ... more convs as you find is useful
    self.flatten = Flatten()
    self.dense = Dense(32)
    self.means = Dense(latent_dim)
    self.variances = Dense(latent_dim)
```

```
def call(self, x):
  x = tf.nn.relu(self.conv1(x))
  # more convs...
  x = self.flatten(x)
  x = tf.nn.relu(self.dense(x))
  means = self.means(x)
  vars = self.variances(x)
  return means, vars
```

Notice we can return multiple values from the call method (as usual, when using the subclassing API, you can write the forward pass imperatively).

Decoder

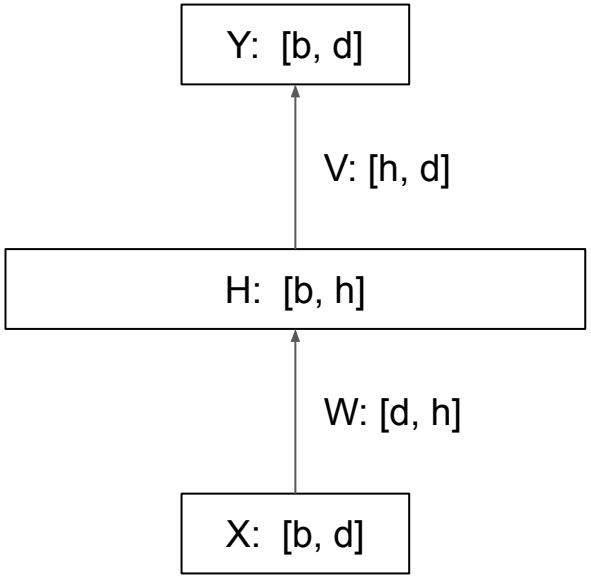
Embedding space -> image space

```
class Decoder(tf.keras.Model):
  def __init__(self):
    super(Decoder, self).__init__()
    self.dense1 = Dense(units=14*14*64, activation=tf.nn.relu)
    self.reshape = Reshape(target_shape=(14, 14, 64))
    self.conv_transpose = Conv2DTranspose(64, 3,strides=(2, 2),padding='same')
    self.conv1 = Conv2D(filters=1, kernel_size=3, padding='same')

def call(self, x):
  x = self.dense1(x)
  x = self.reshape(x)
  x = tf.nn.relu(self.conv_transpose(x))
  return tf.nn.sigmoid(self.conv1(x))
```

Data and model parallelism

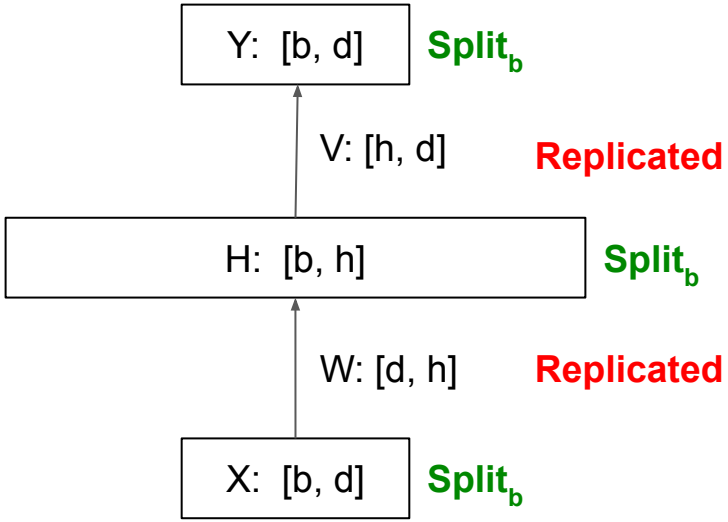
Example Perceptron

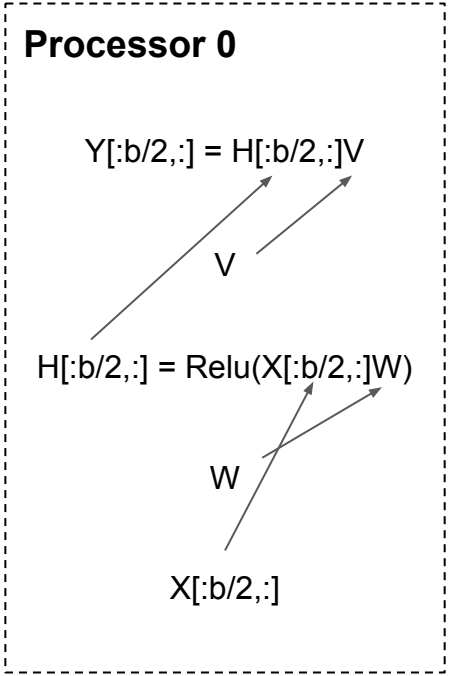
$$Y = (H = \text{Relu}(XW))V$$

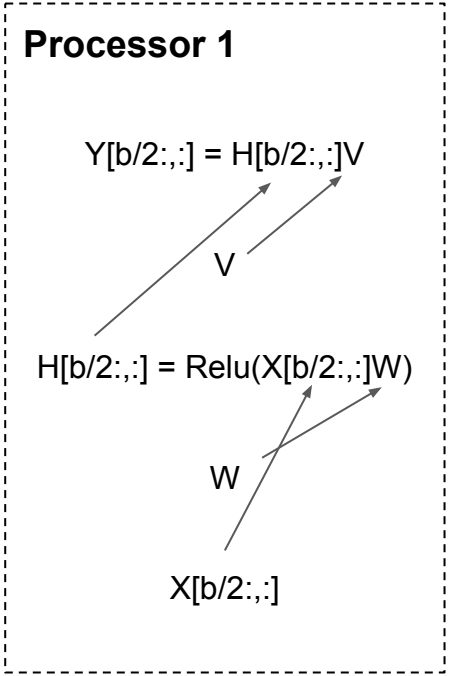


Data Parallelism: split dimension “b”

Y = (H = Relu(XW))V

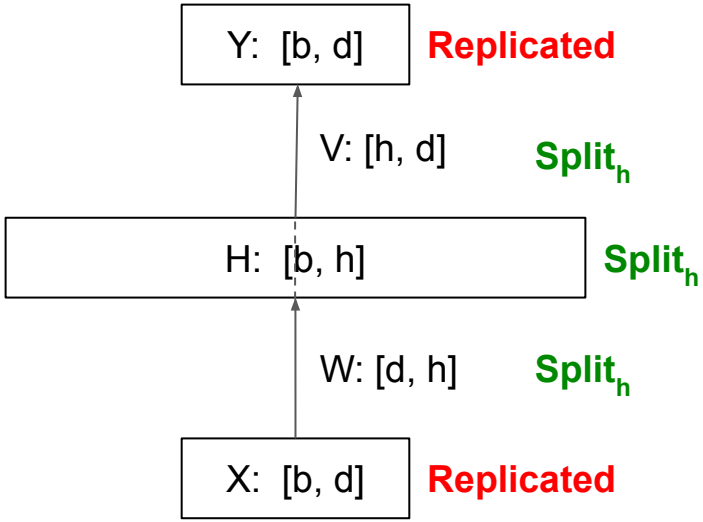


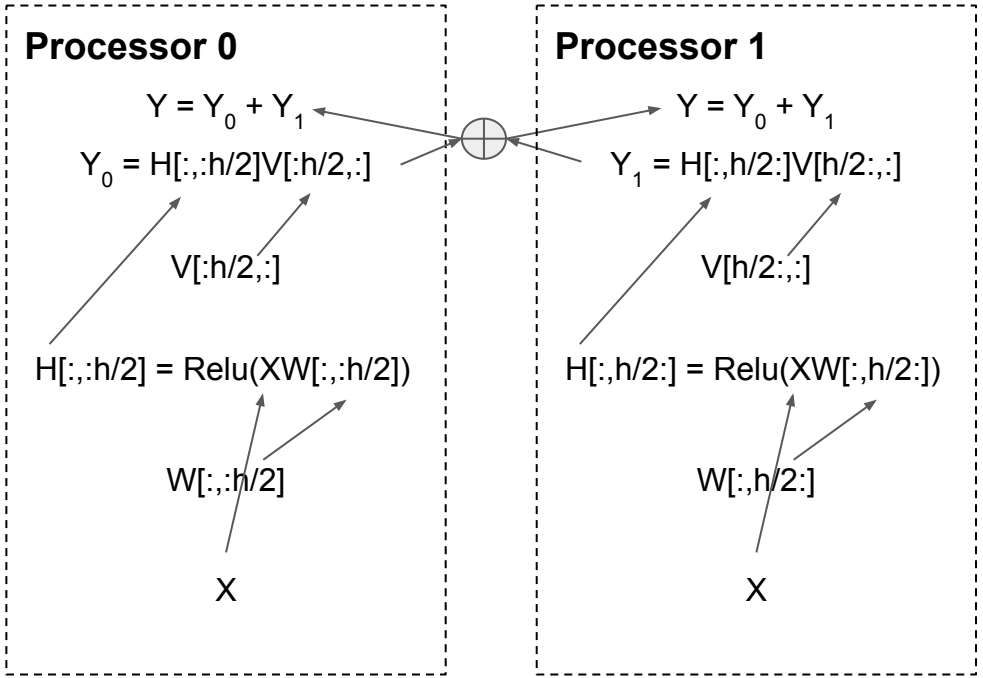




Model Parallelism: split dimension “h”

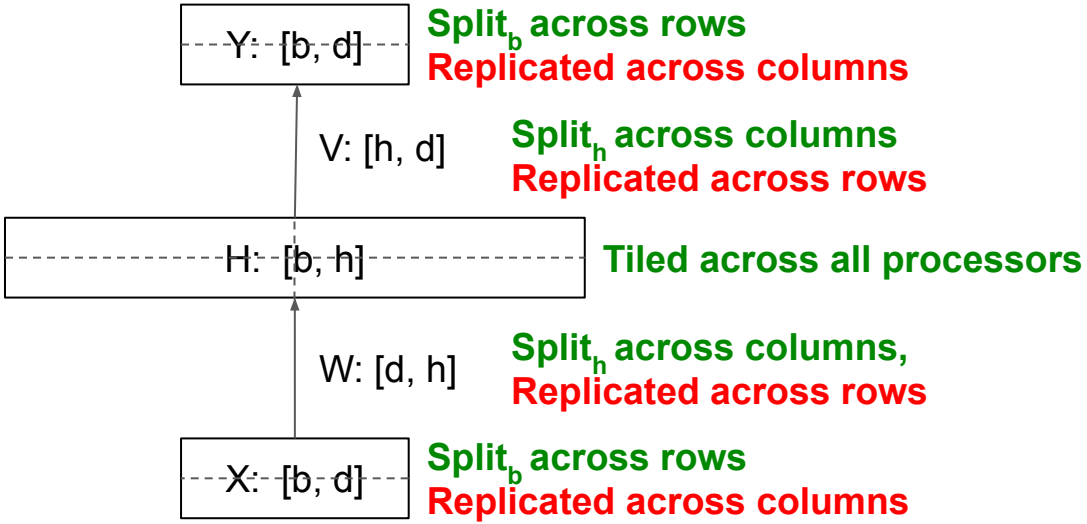
$Y = (H = Relu(XW))V$

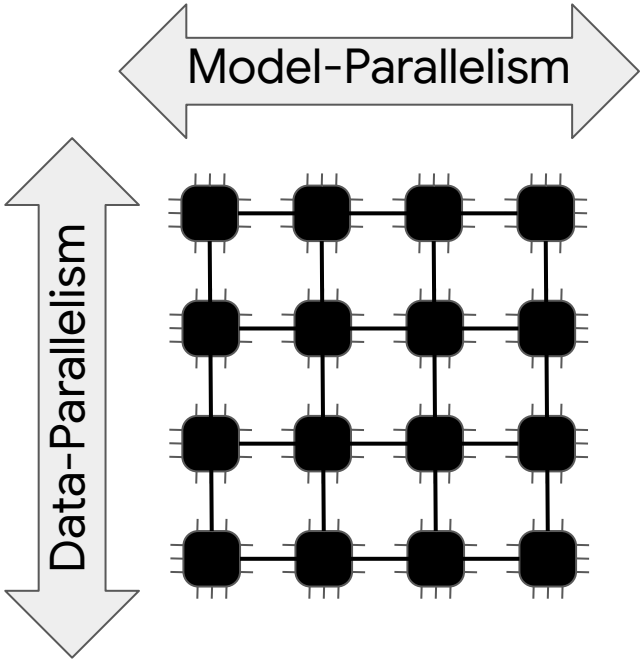




Data and Model Parallelism on 2D Mesh

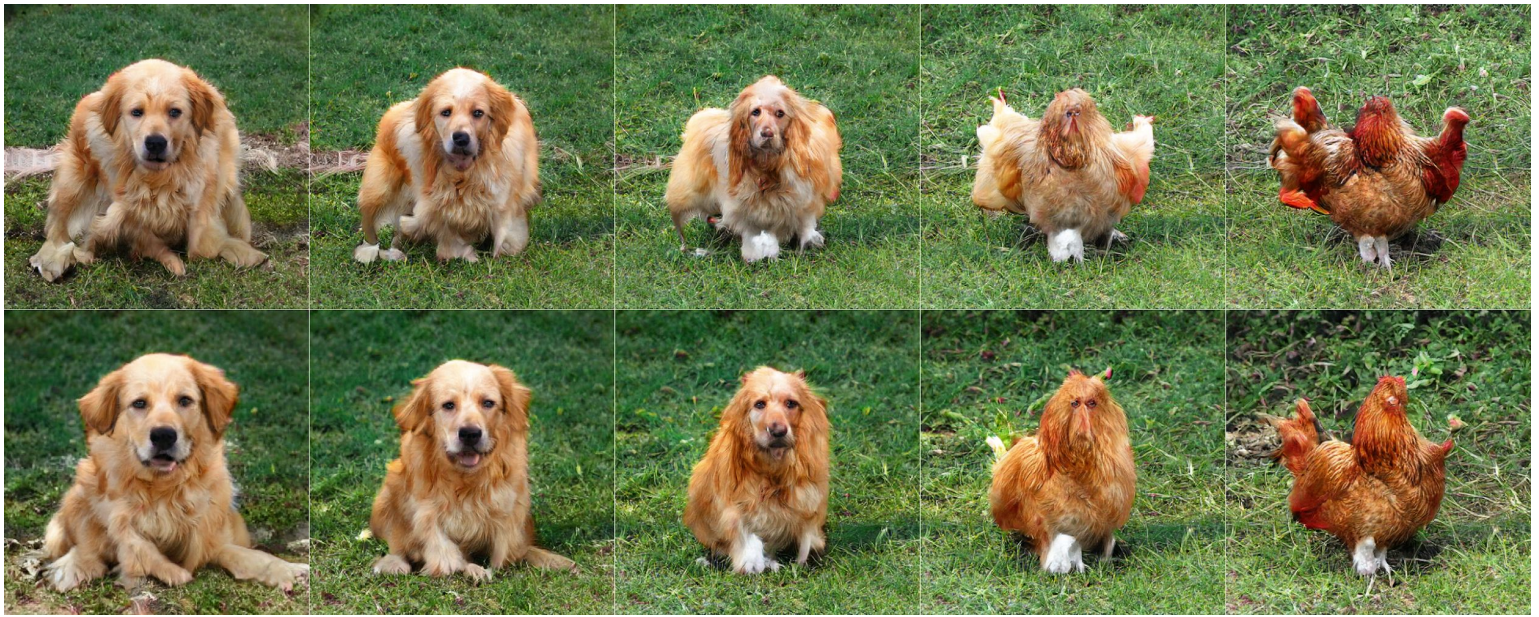
Split batch “b” across rows of processors, “h” across columns of processors





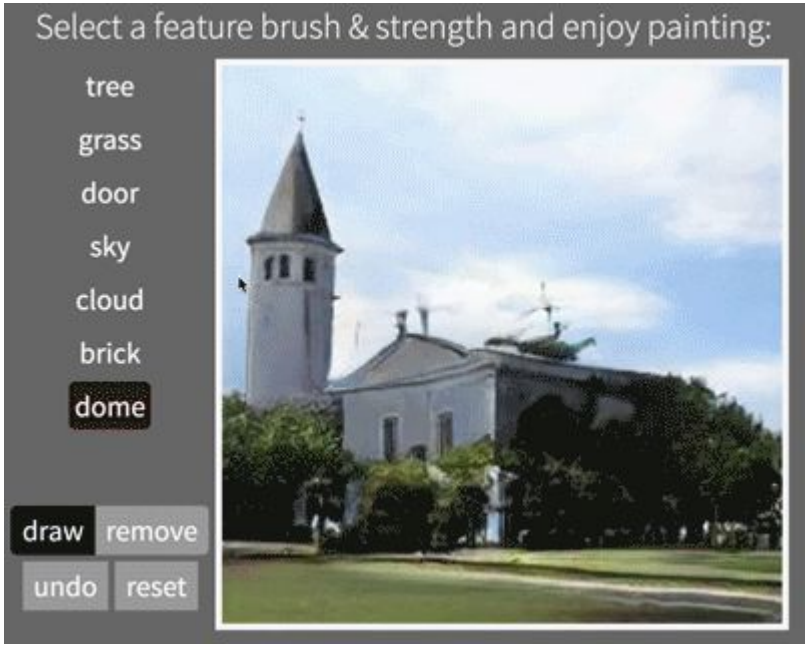
Finally, two awesome GAN papers.

BigGAN - Try it!



Large Scale GAN Training for High Fidelity Natural Image Synthesis

GAN Dissection - Try it!



Historically speaking, VAEs have been more appealing for artistic applications, because they're easier to directly control, though this may have changed a few weeks ago with this work from MIT.

https://gandissect.csail.mit.edu/

Reading and learning more

Reading

Generative Adversarial Nets, 2014

CycleGan (Unpaired Image-to-Image Translation), 2017

Learning more

Aurélien Geron has nearly finished his book on TensorFlow 2.0.