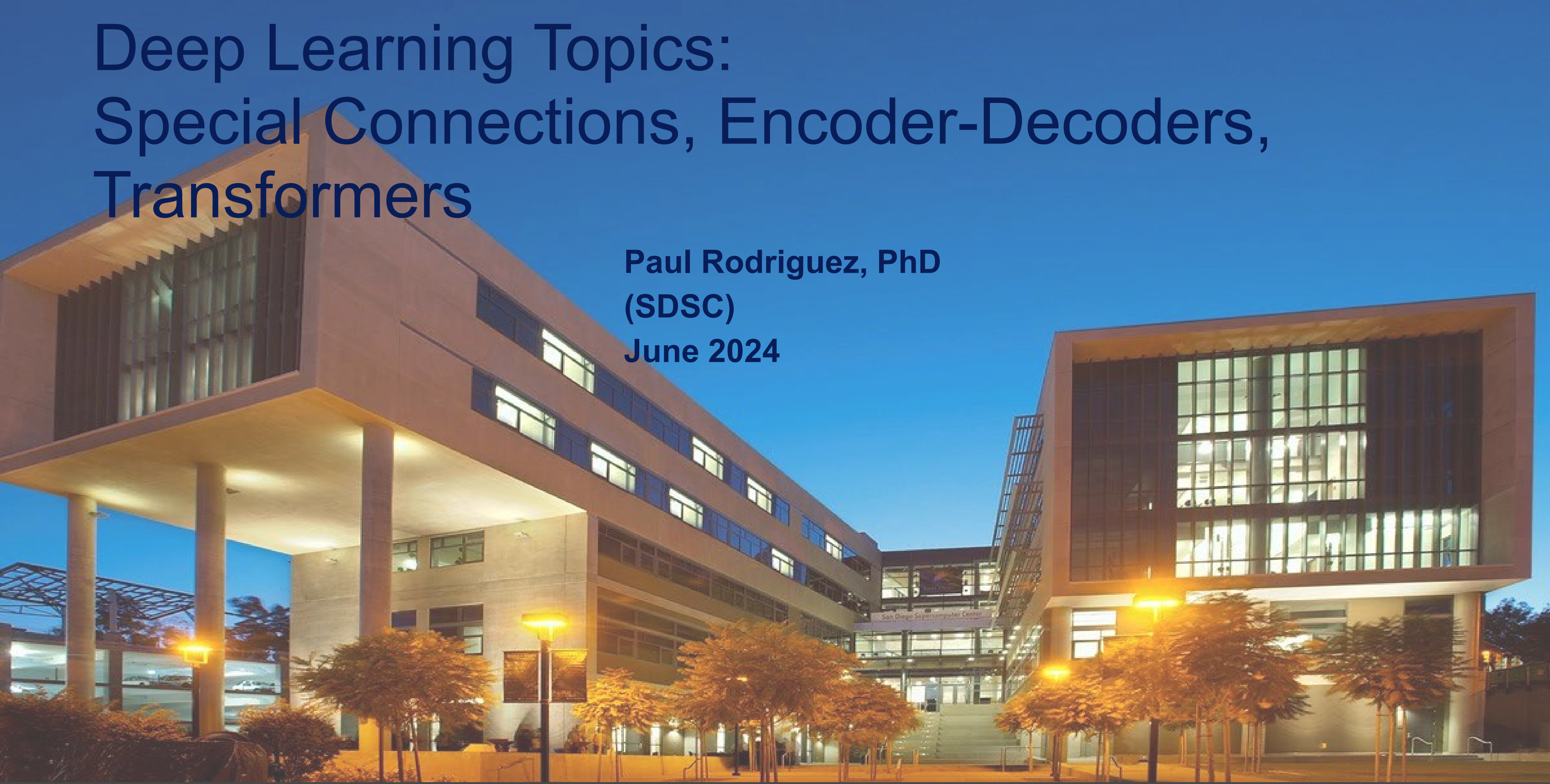
Deep Learning Topics: Special Connections, Encoder-Decoders, Transformers

Paul Rodriguez, PhD
(SDSC)
June 2024



Outline

- **Part I**

Gate connection idea

Skip and Residual connections

Programing connections and Keras Model API

Encoder-Decoder (Autoencoder)

Exercise MNIST Autoencoder

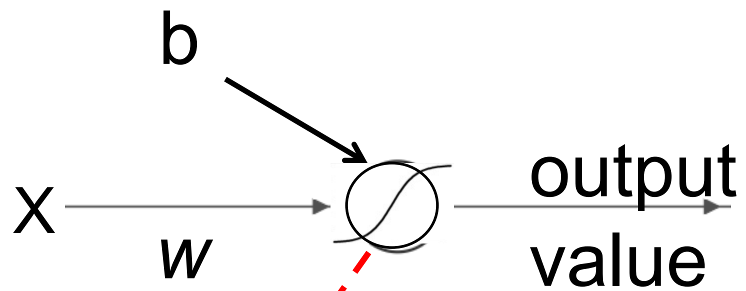
Autoencoder with Stable Diffusion

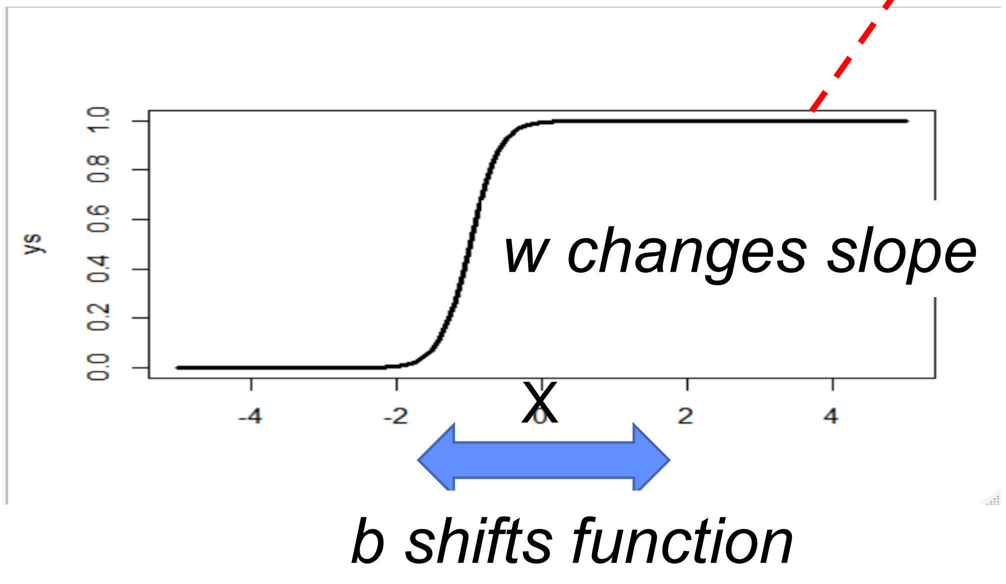
- **Part II**

Attention Head and Transformers,

Recall: the logistic unit

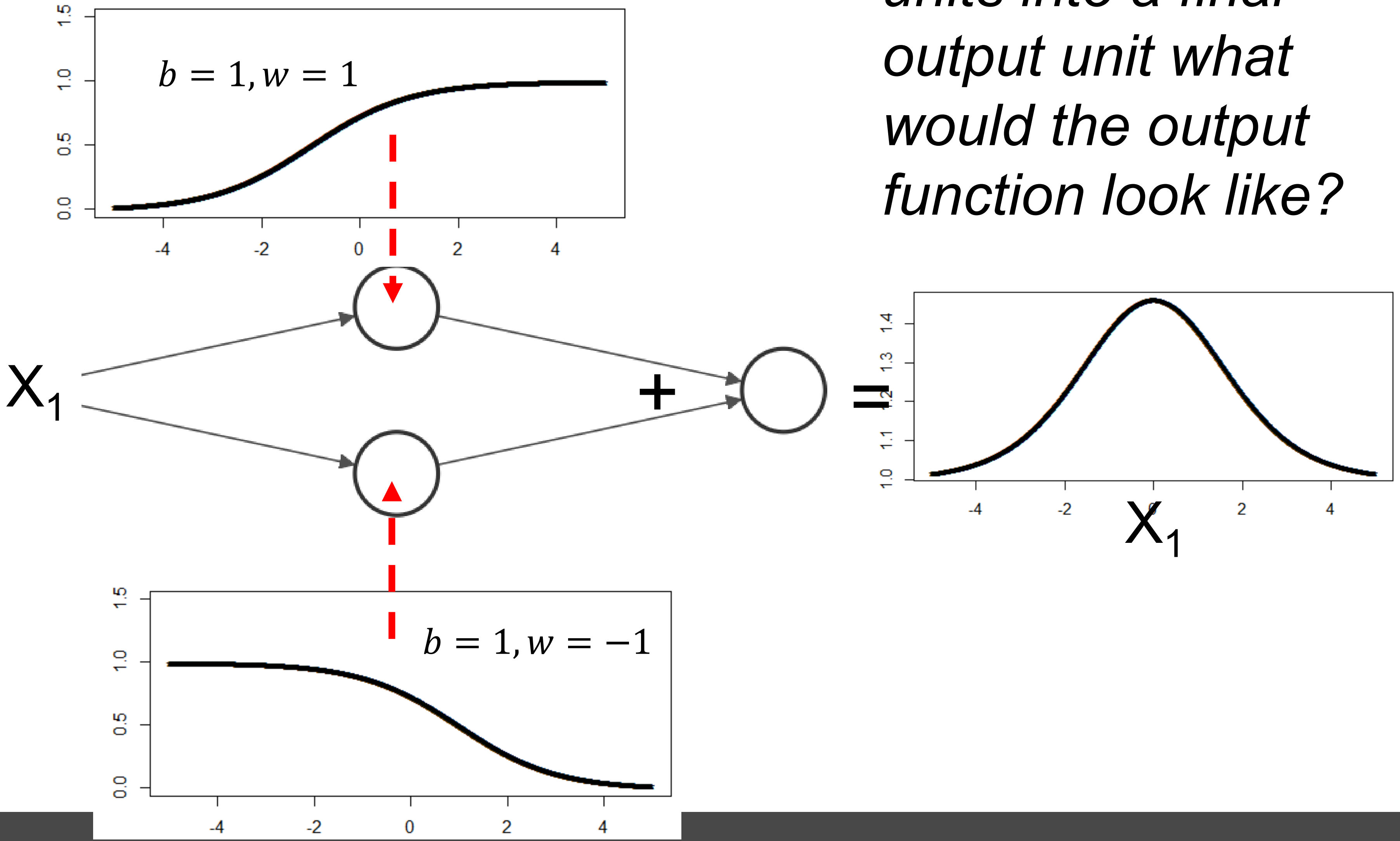
$$f(x) = \frac{1}{1 + exp^{(-(b+wx))}} \Leftrightarrow$$



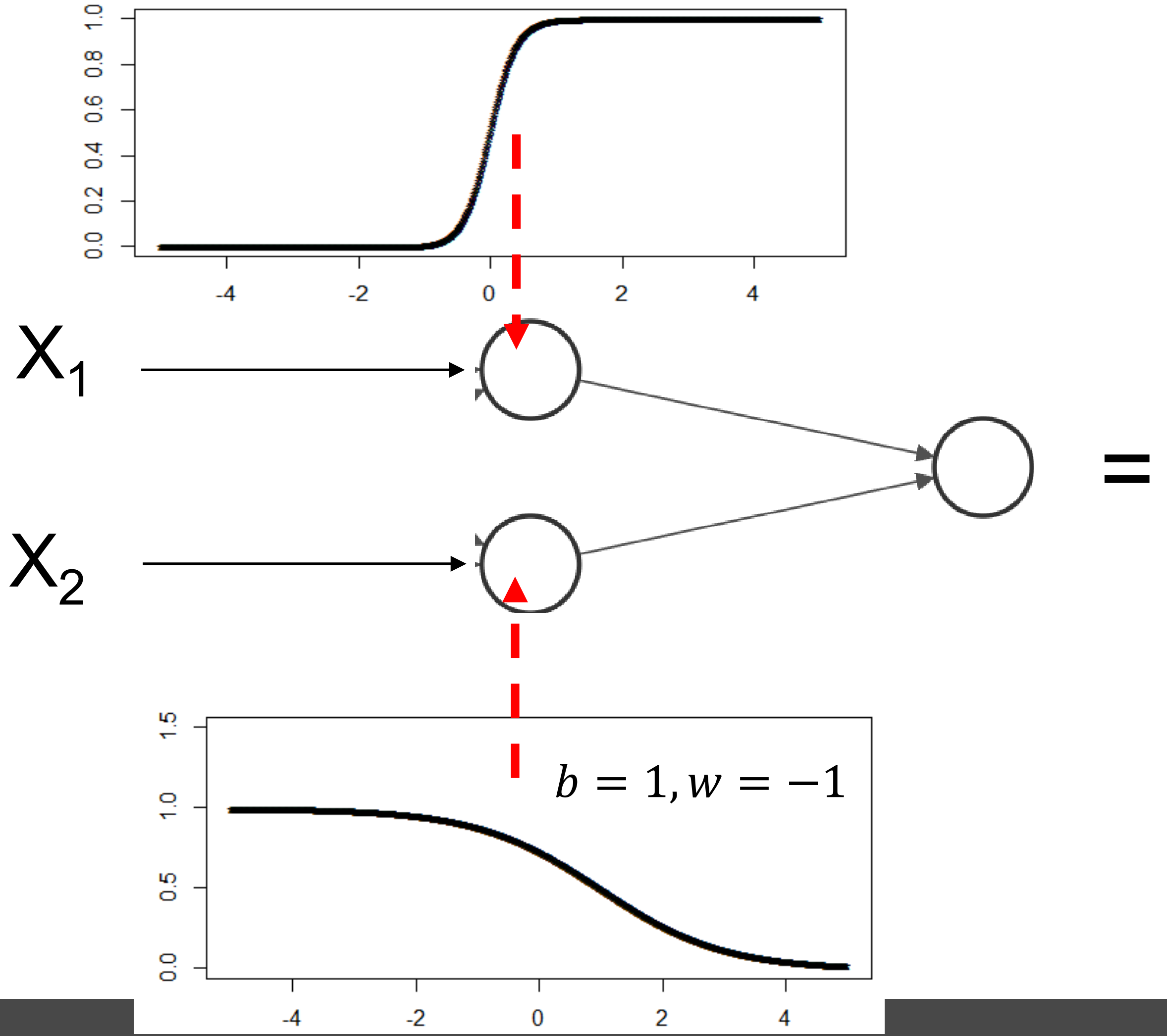


Example: 1 input into 2 logistic units with these activations

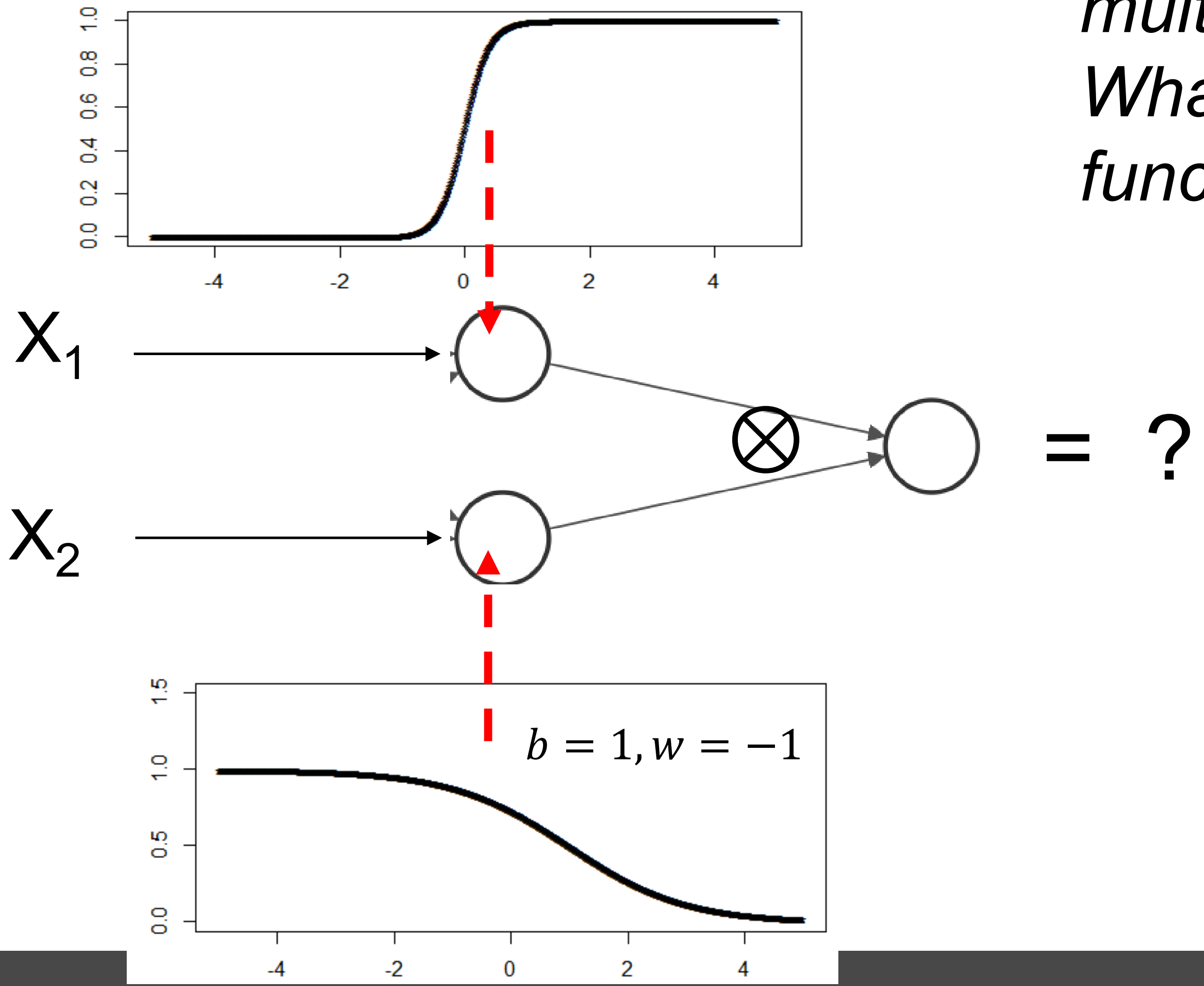
If you add these 2 units into a final output unit what would the output function look like?



Example: 2 input into 2 logistic units with these activations

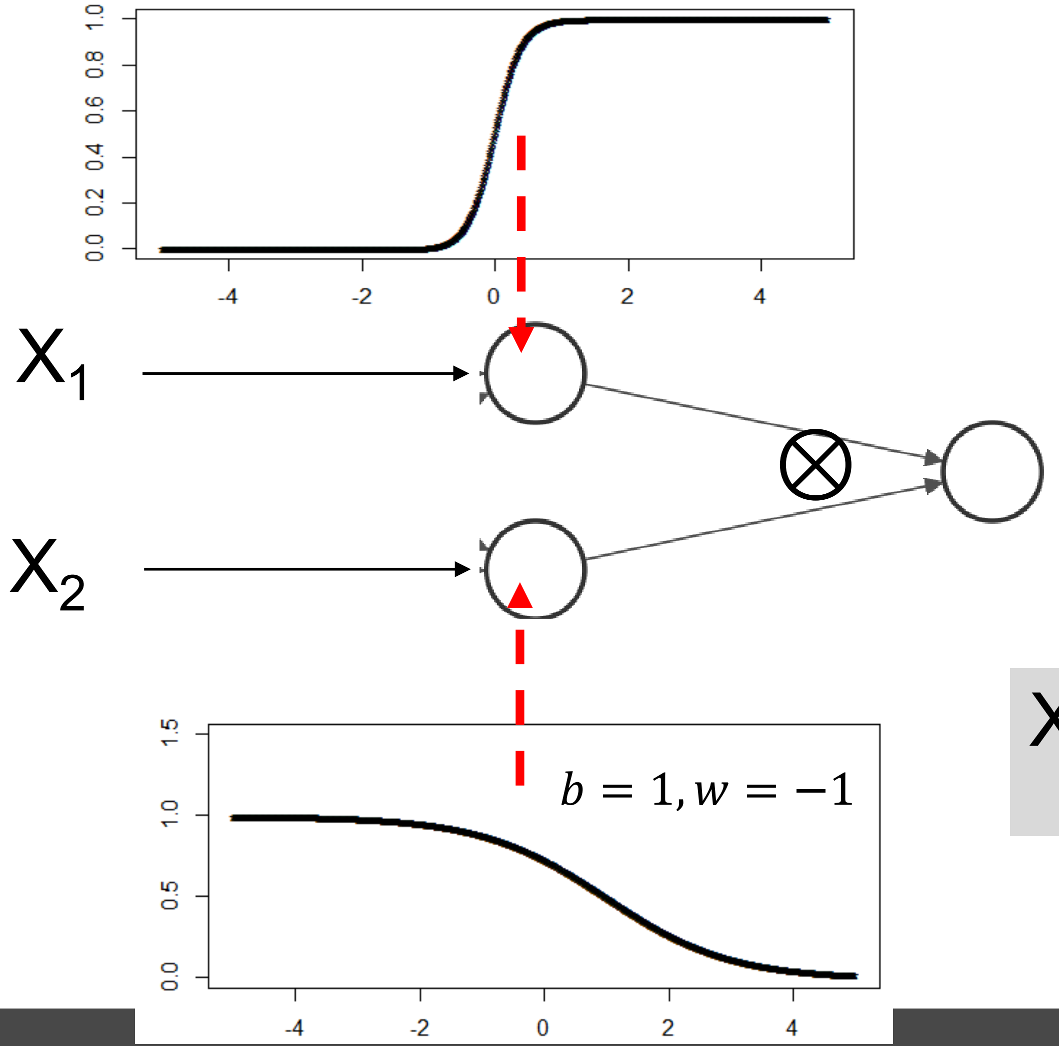


Example: 2 input into 2 logistic units with these activations



What if you multiply these? What is the output function doing?

Example: 2 input into 2 logistic units with these activations

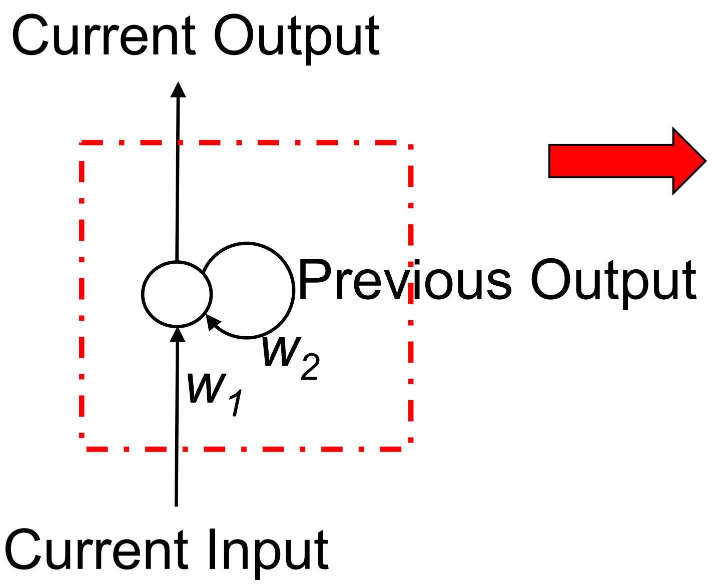


What if you multiply these?
For linear activation, what is the output function doing ?

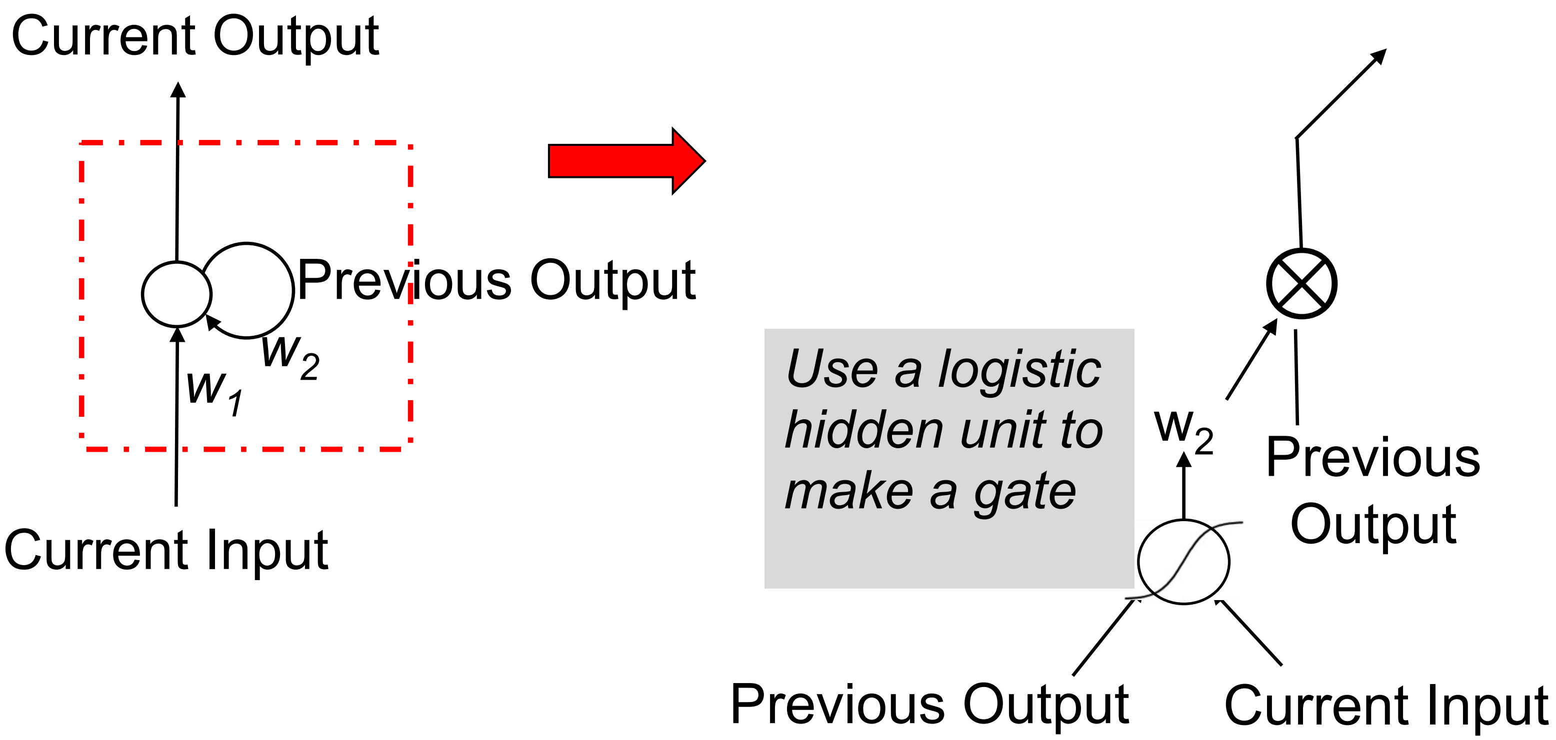
$$= \begin{cases} 0 & \text{if } X_1 < 0 \\ h(X_2) & \text{if } X_1 > 0 \end{cases}$$

X_1"gates" X_2 activation

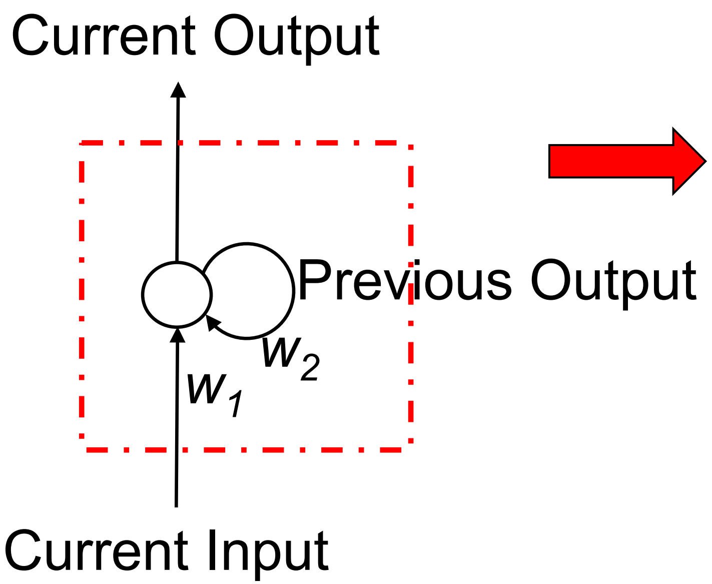
A recurrent unit for sequence learning can be replaced by a gated unit

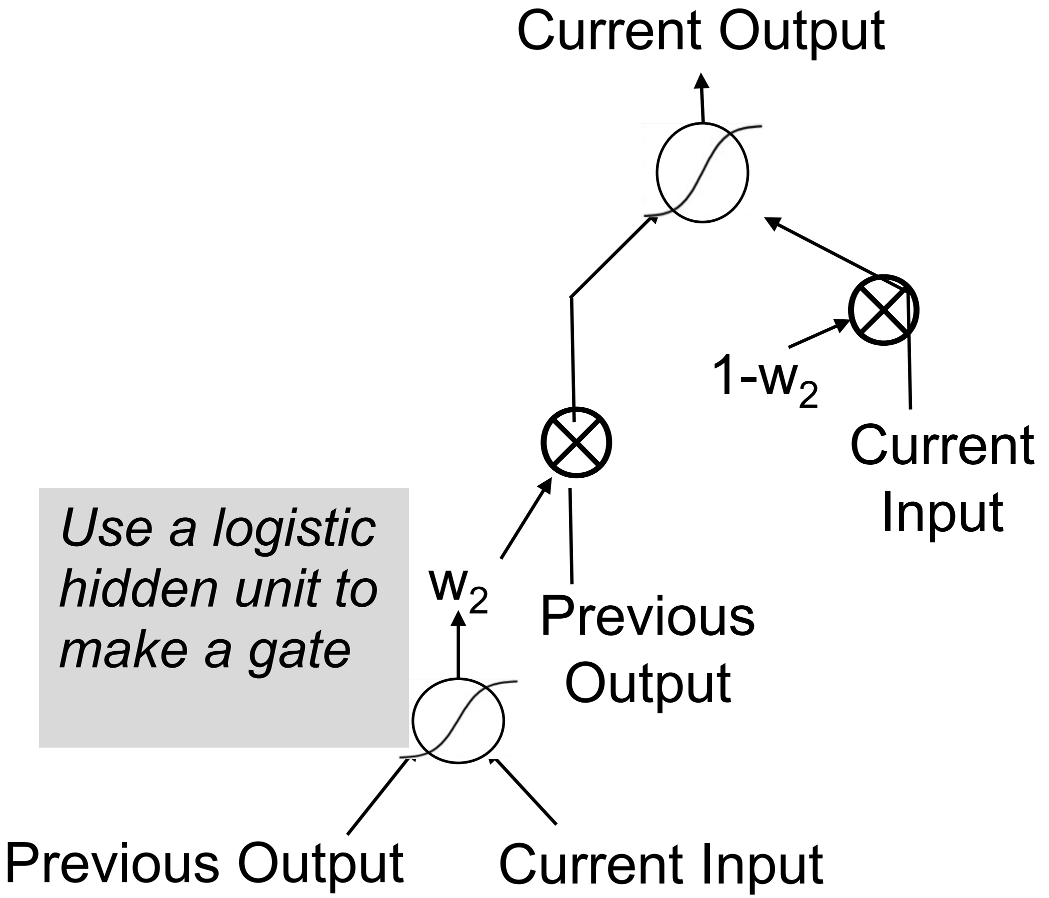


A recurrent unit for sequence learning can be replaced by a gated unit



A recurrent unit for sequence learning can be replaced by a gated unit



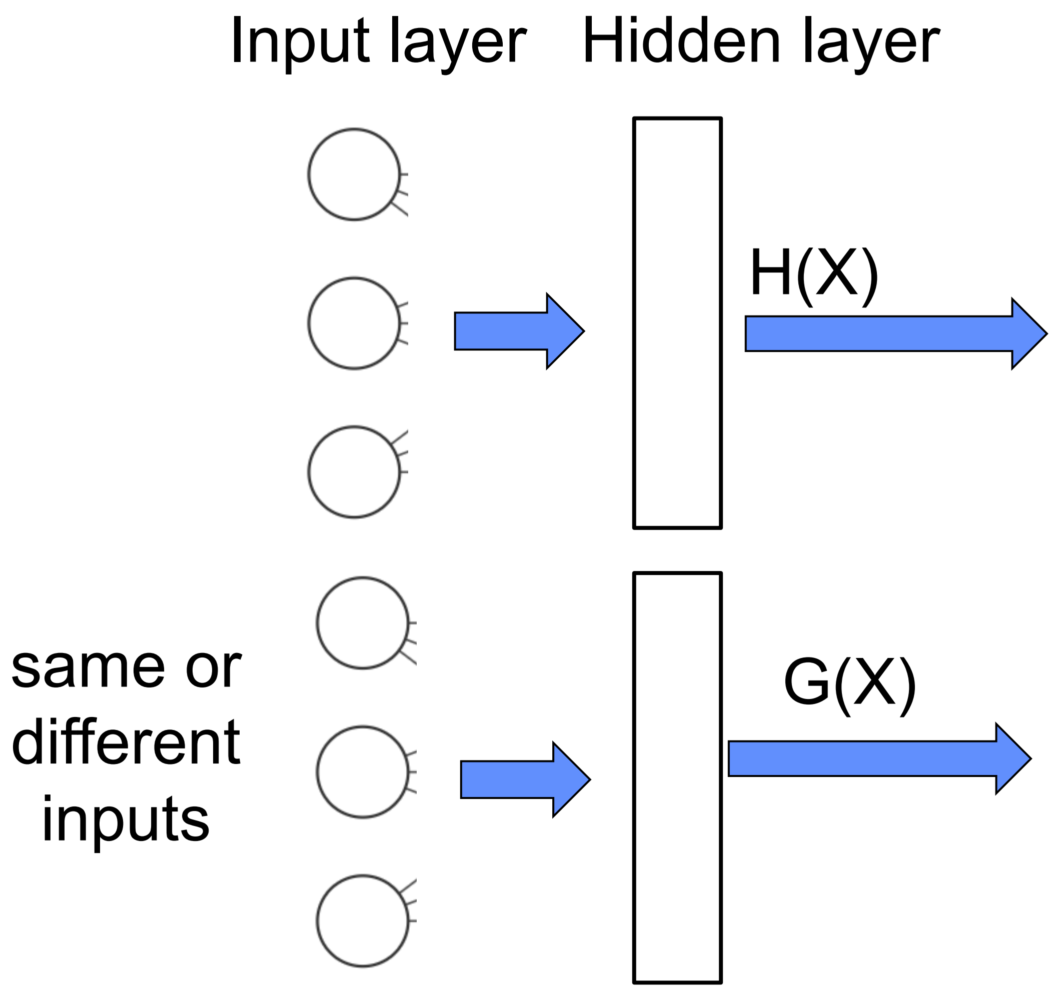


Use the gate to either keep previous output or update it with current input

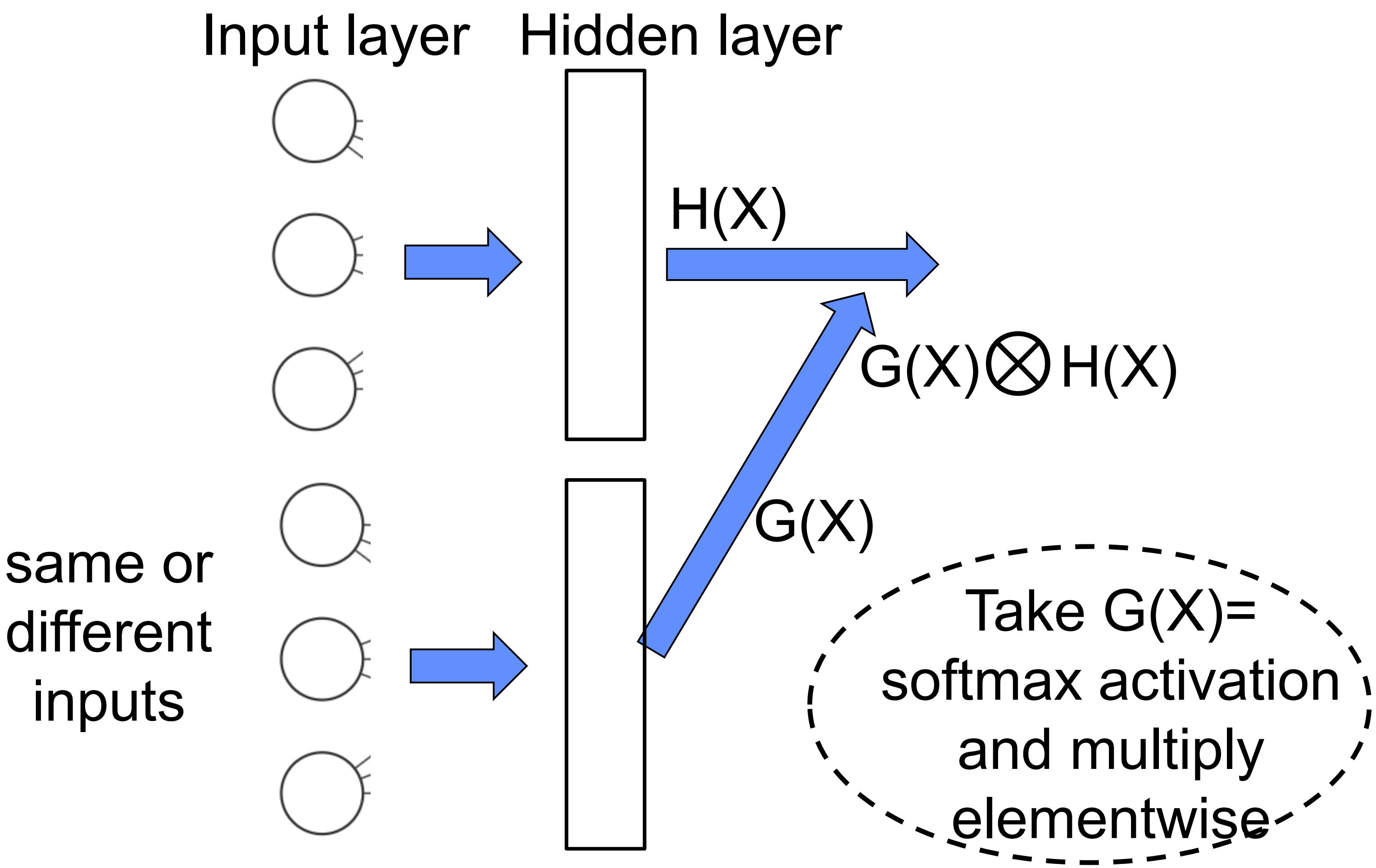
Use a logistic hidden unit to make a gate

'Gated Recurrent Unit'
Cho, Bengio 2015

Redrawing the gate for two sets of hidden units



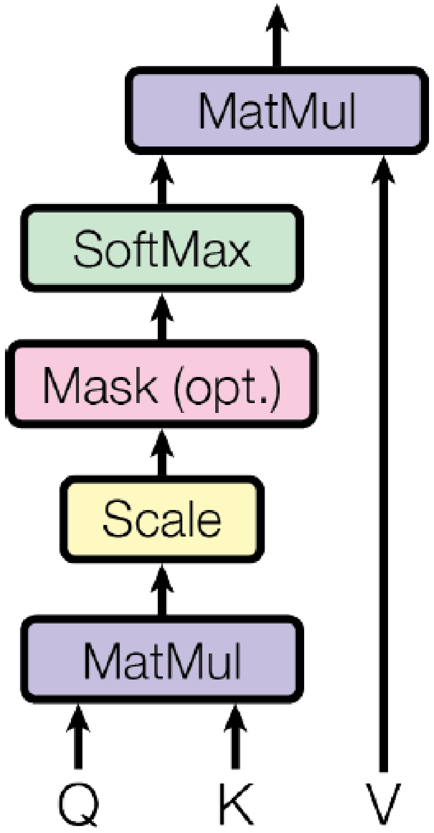
Use softmax for G(X) to get gating weights



Recall: softmax normalizes outputs into probability weights

Scaled Dot-Product Attention
(very rough summary)

"Attention" mechanism in language transformers use a softmax gate

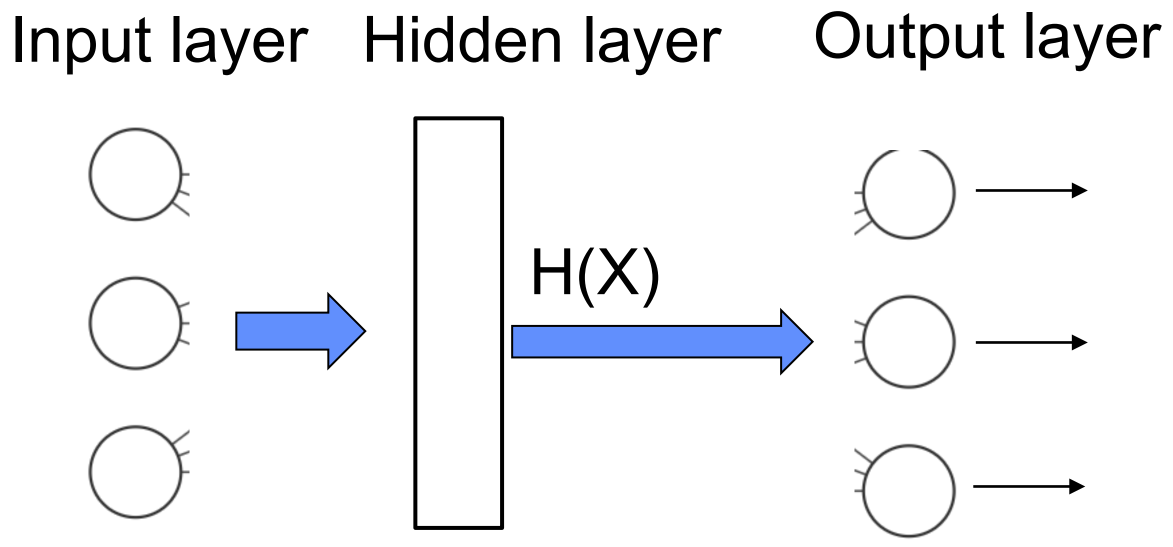


The gate is applied to possible Values (V) for decoding

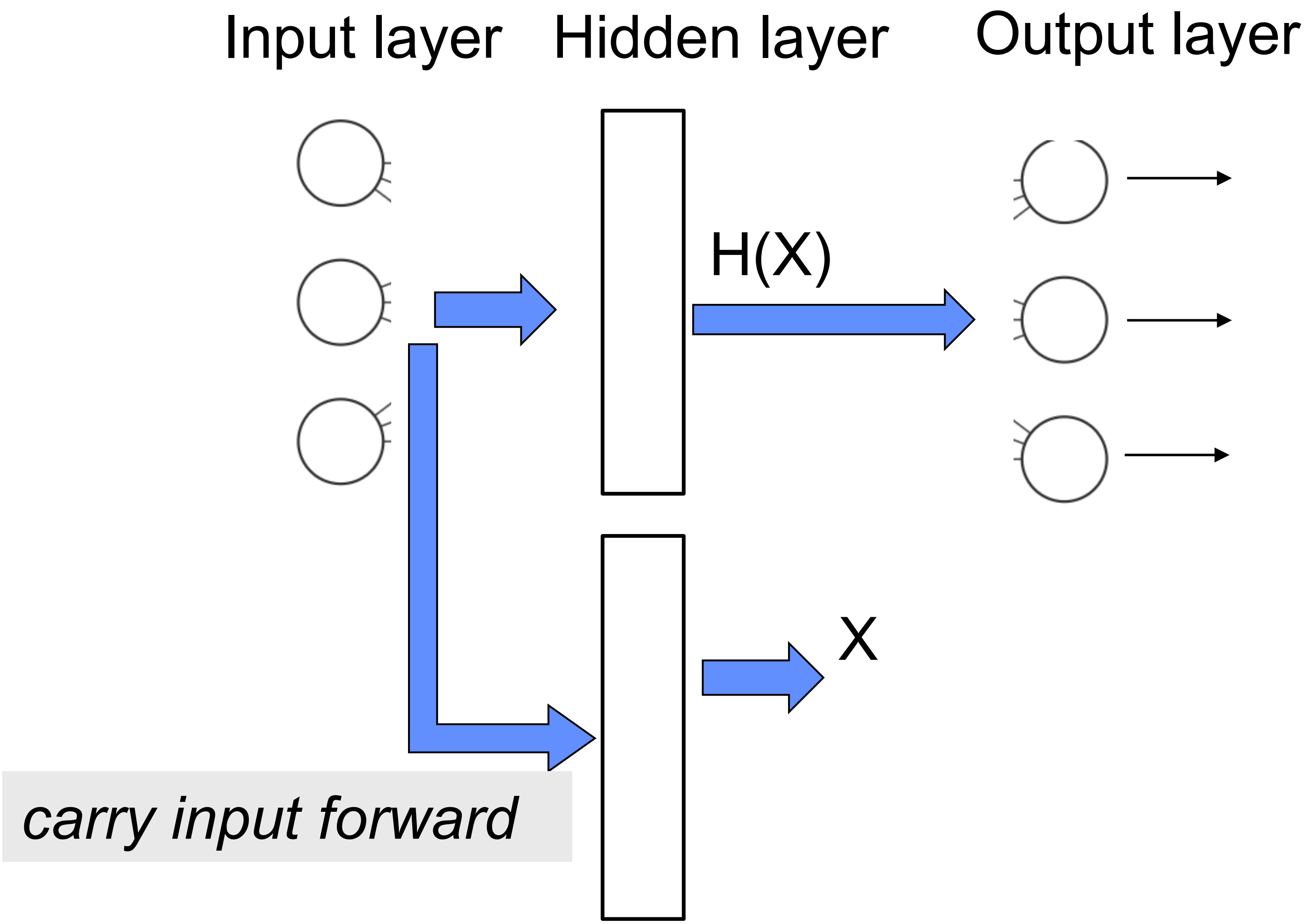
Q,K,V depend on input

Vaswani, et al. 2017
Attention Is All You Need (for Transformers)

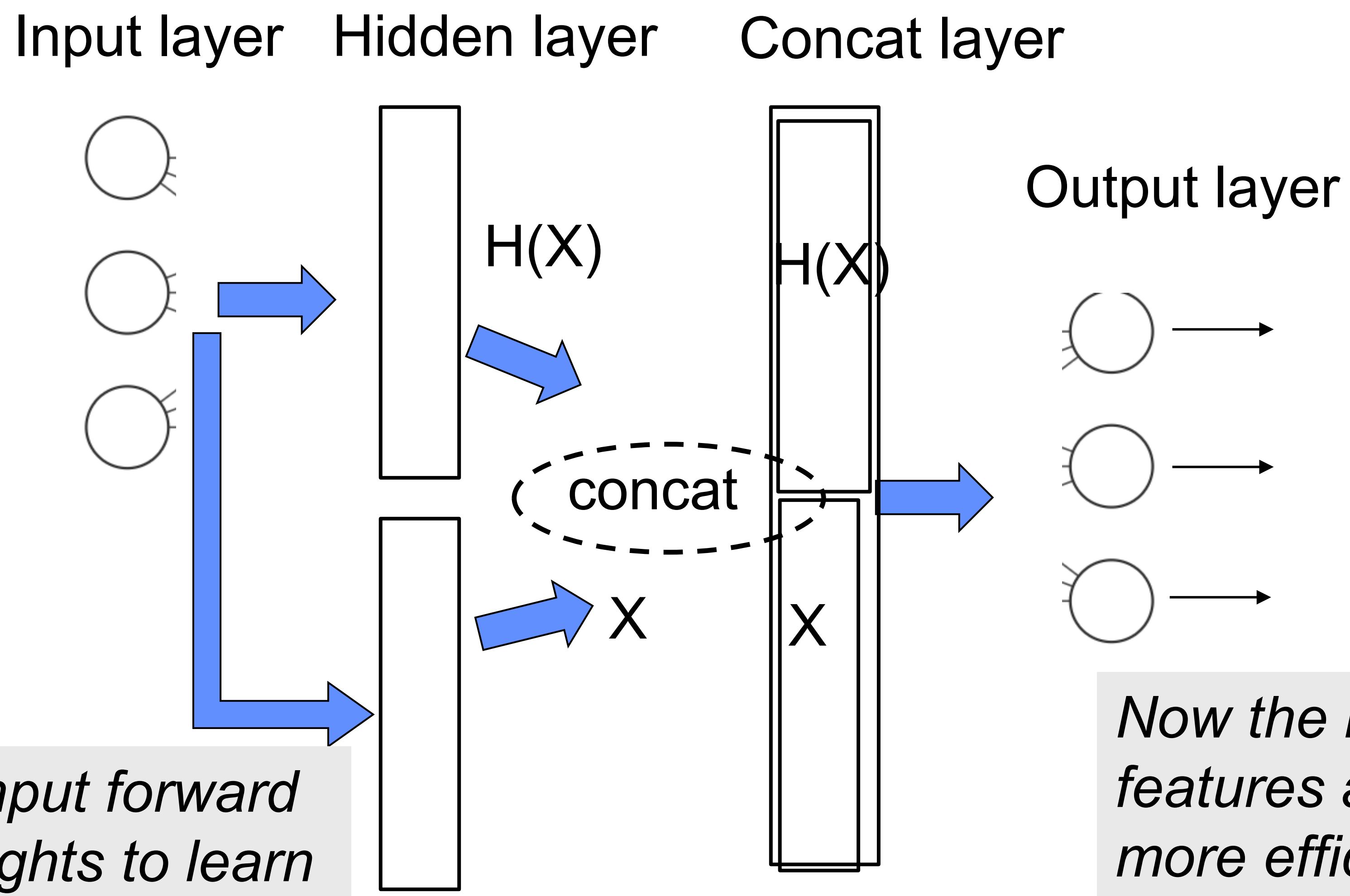
Recall the Multilayer Perceptron (MLP)



To help the MLP learn directly from input carry input forward



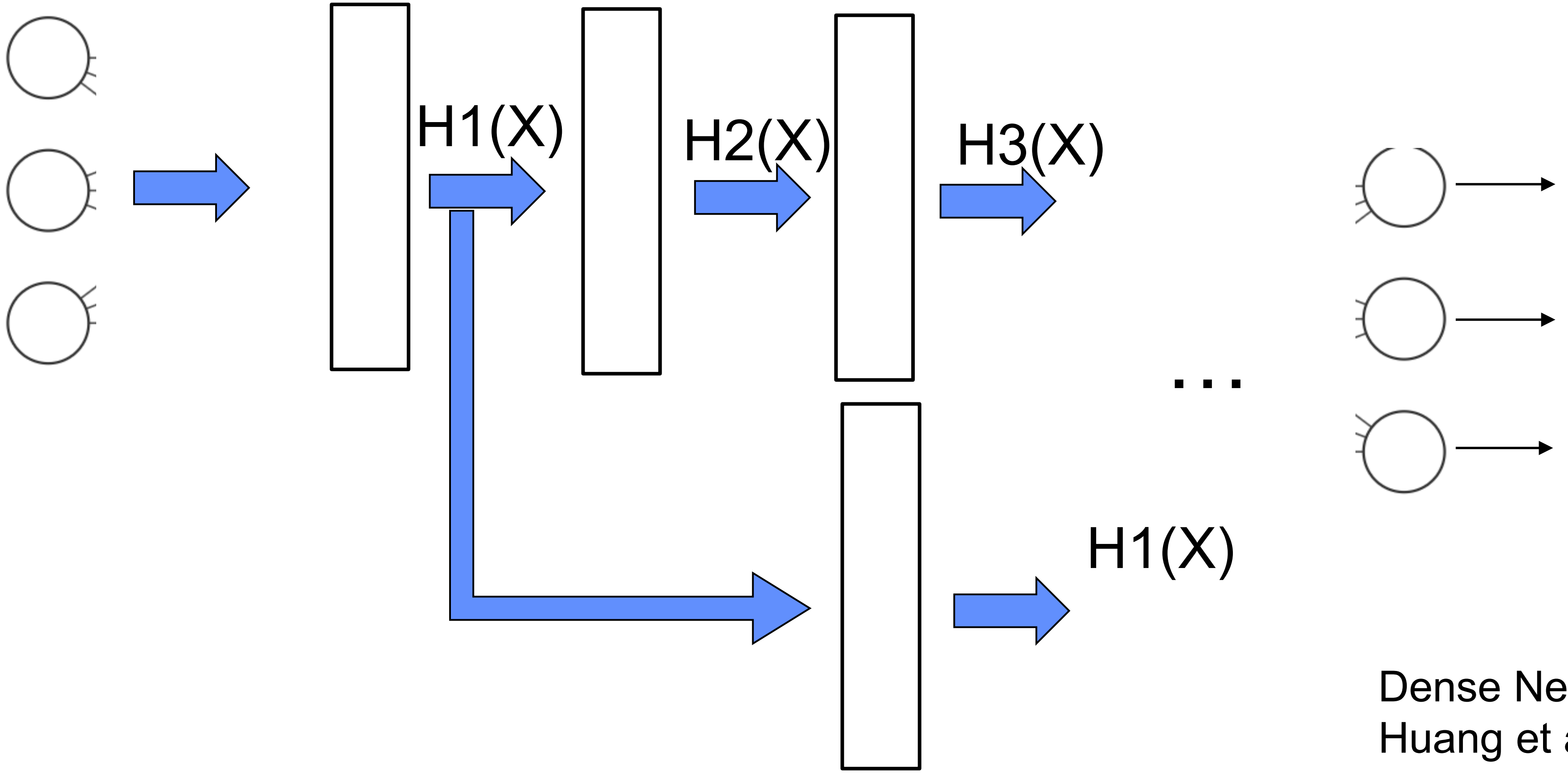
Concatenate input with hidden units into new layer



carry input forward
No weights to learn

Now the input features are used more efficiently

Can be done for any (or all) previous layer and *skip* any number of layers

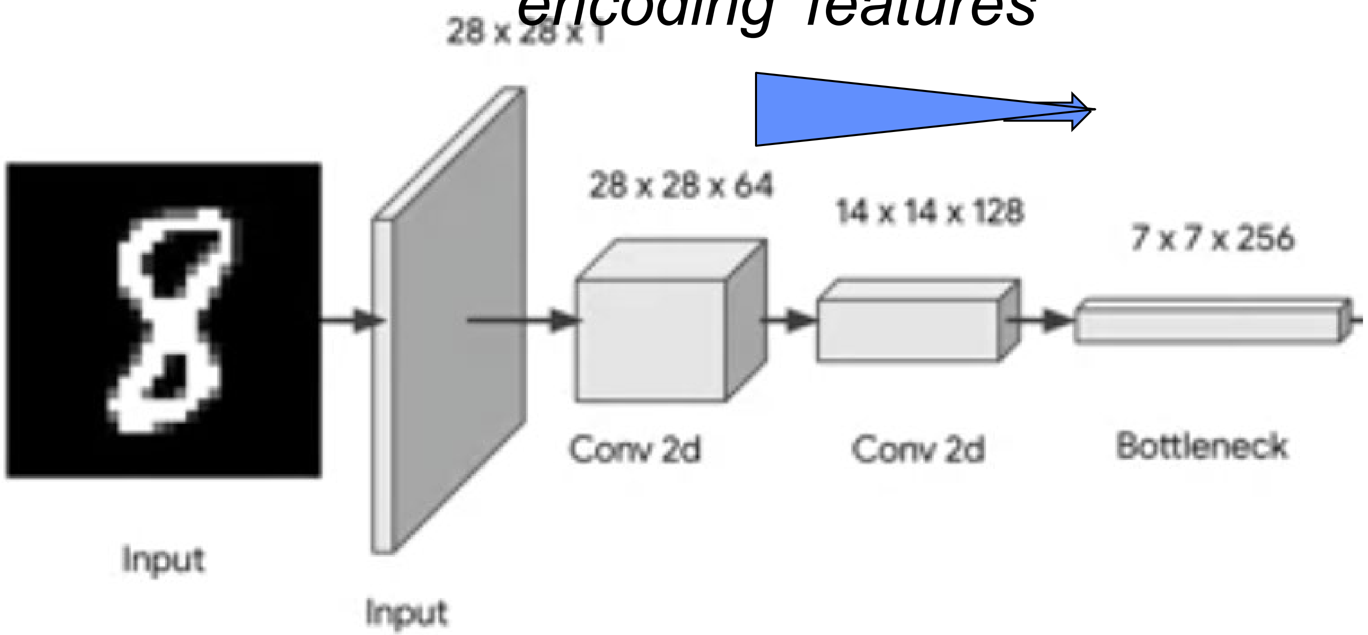


Dense Nets
Huang et al

Recall: CNN architecture for MNIST classification

ENCODER

more feature maps & downsampling : 'encoding' features

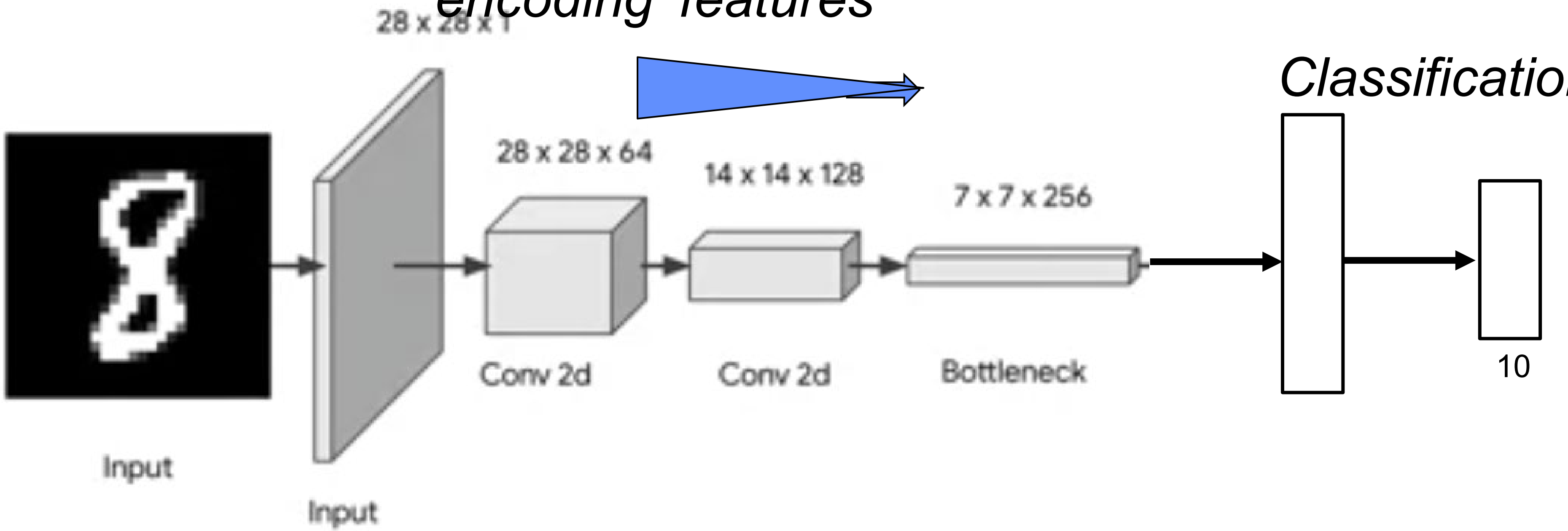


Consider: CNN architecture for MNIST classification

ENCODER

more feature maps & downsampling : 'encoding' features

Classification layers

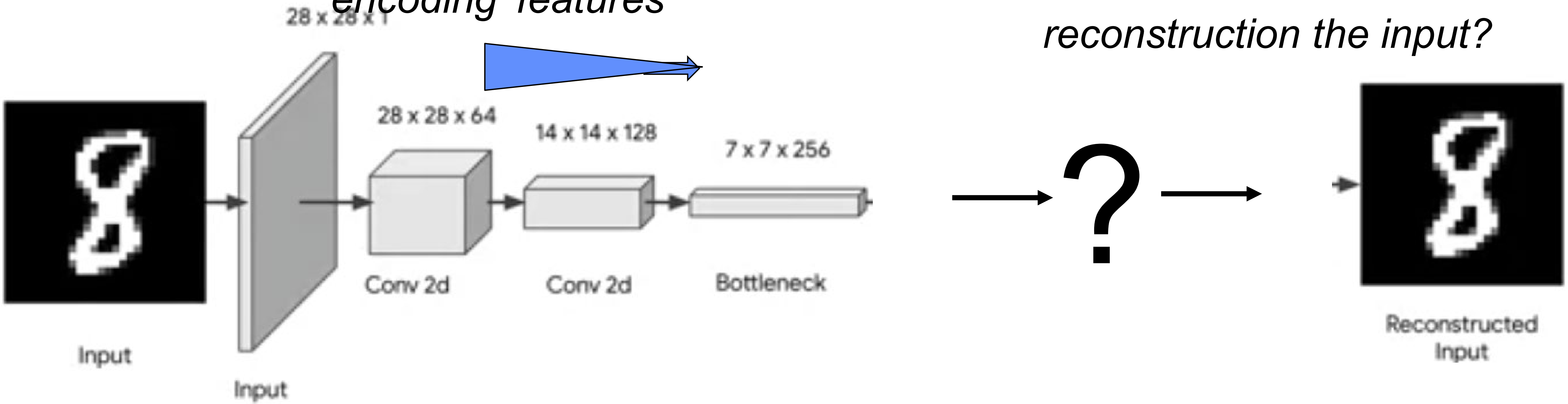


A CNN architecture for MNIST autoencoding

ENCODER

more feature maps & downsampling :
'encoding' features

What if we want to reconstruction the input?



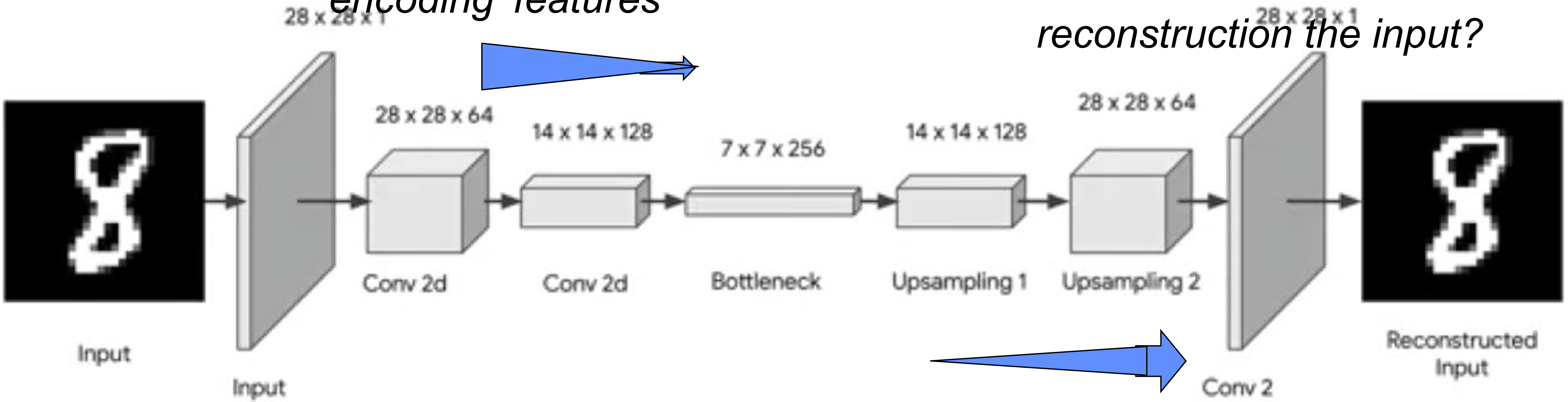
A CNN architecture for MNIST autoencoding

ENCODER

DECODER

more feature maps & downsampling : 'encoding' features

What if we want to reconstruction the input?



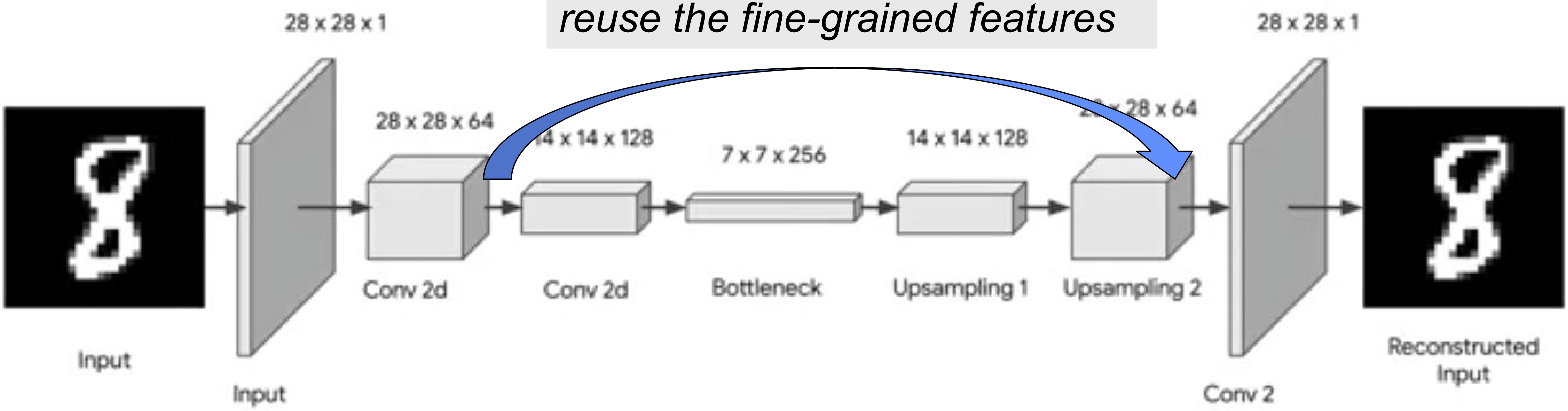
Use 'upsampling' to reverse the encoding – ie. decoding

A CNN architecture for MNIST autoencoding

ENCODER

DECODER

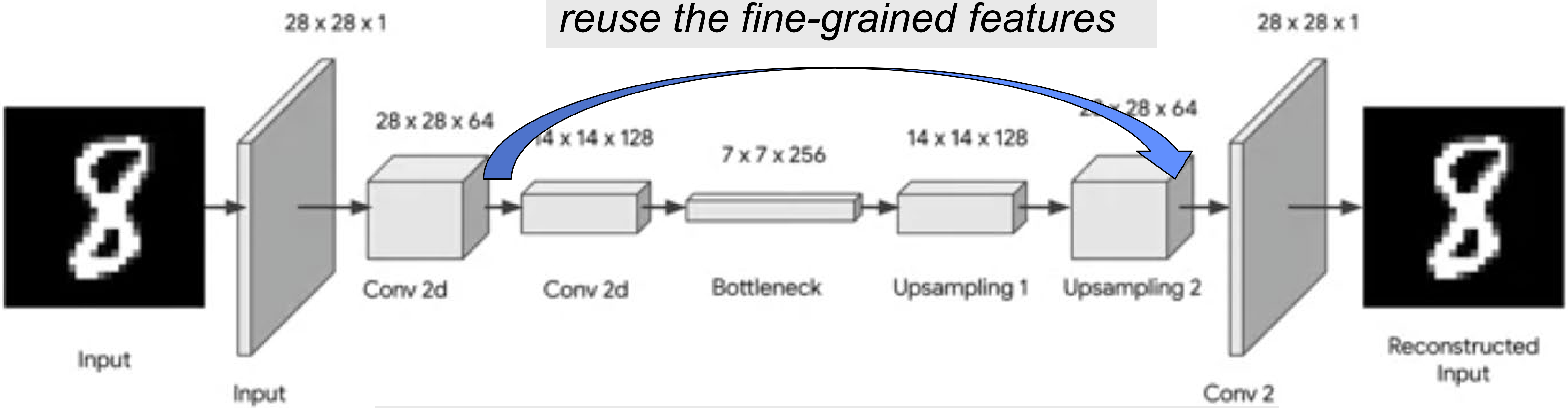
Adding skip connections helps reuse the fine-grained features



A CNN architecture for MNIST autoencoding

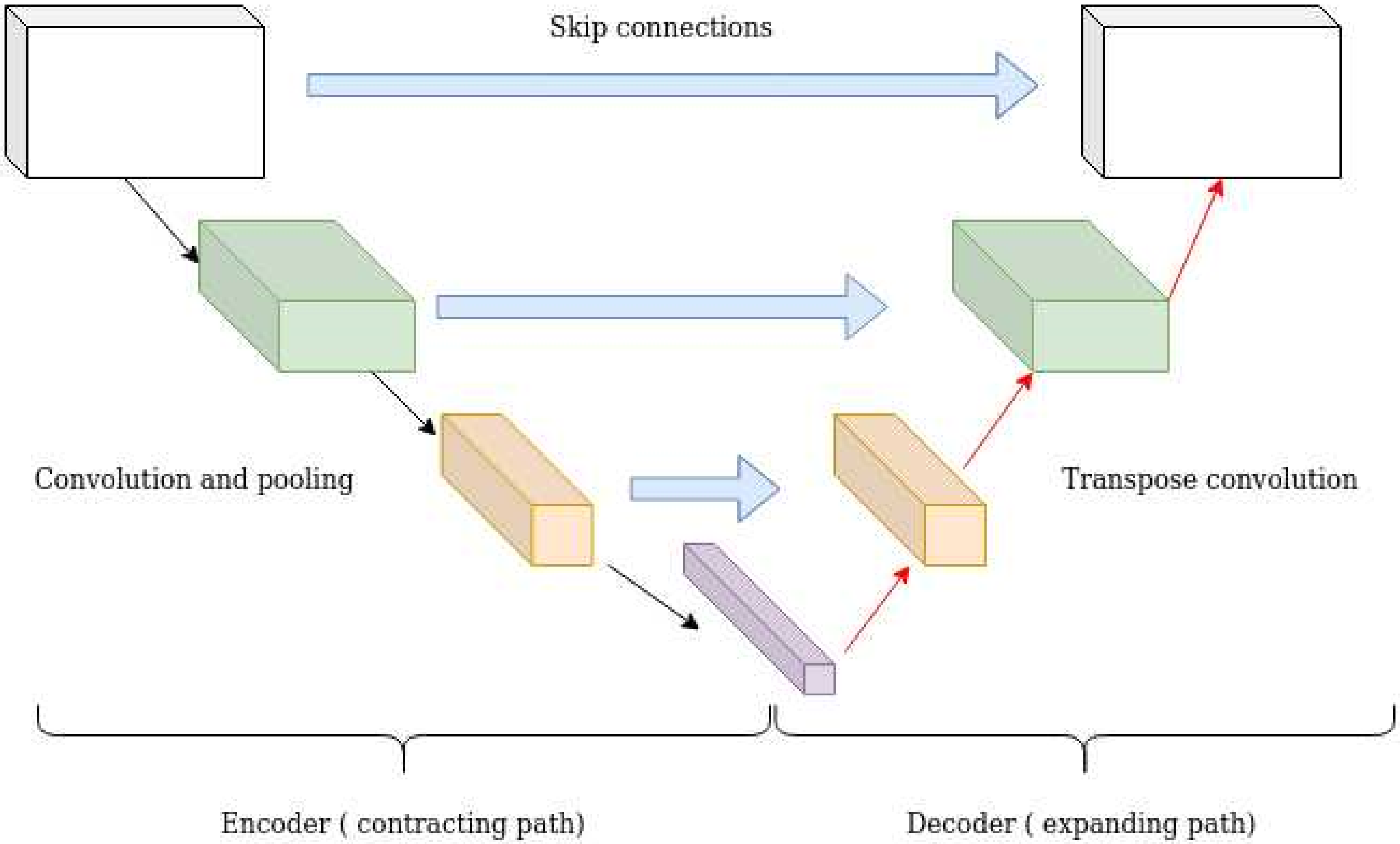
ENCODER *DECODER*

Adding skip connections helps reuse the fine-grained features



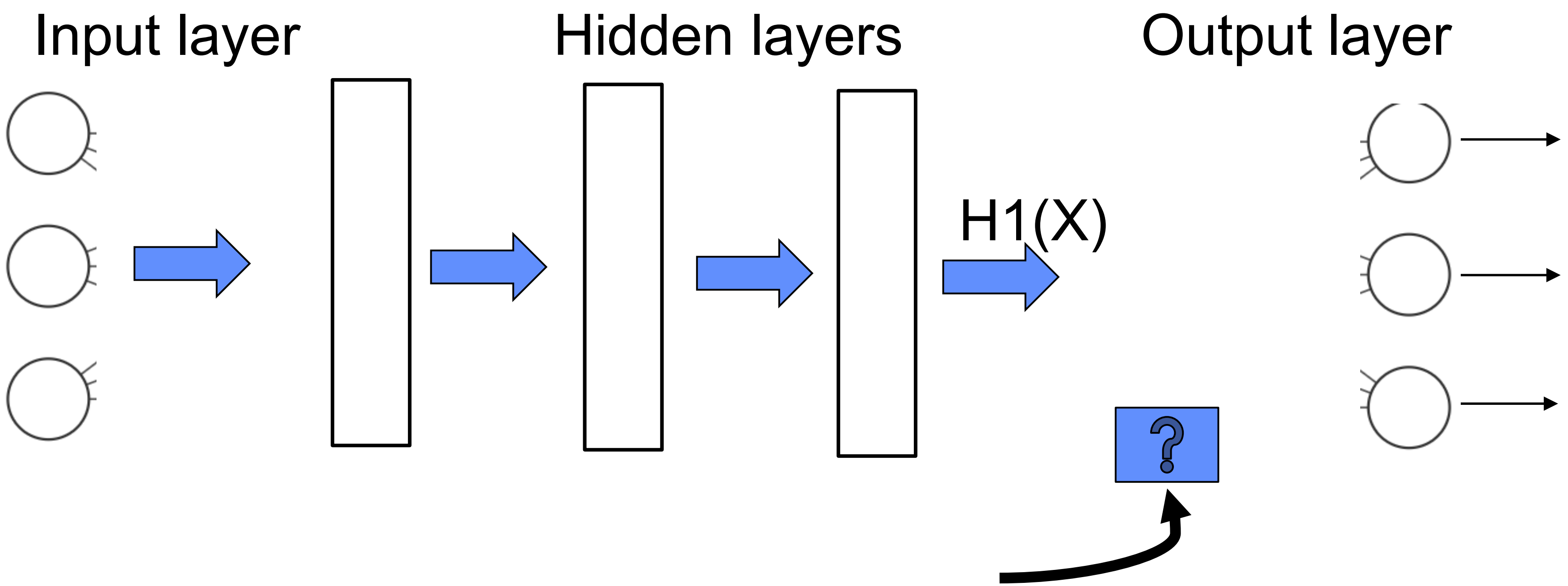
NOTE the 28x28x64 encoded maps have to be skipped ahead to where the 28x28x64 decoding maps are – which axis is concatenated?

Image Encoder-Decoder is a “UNET” architecture



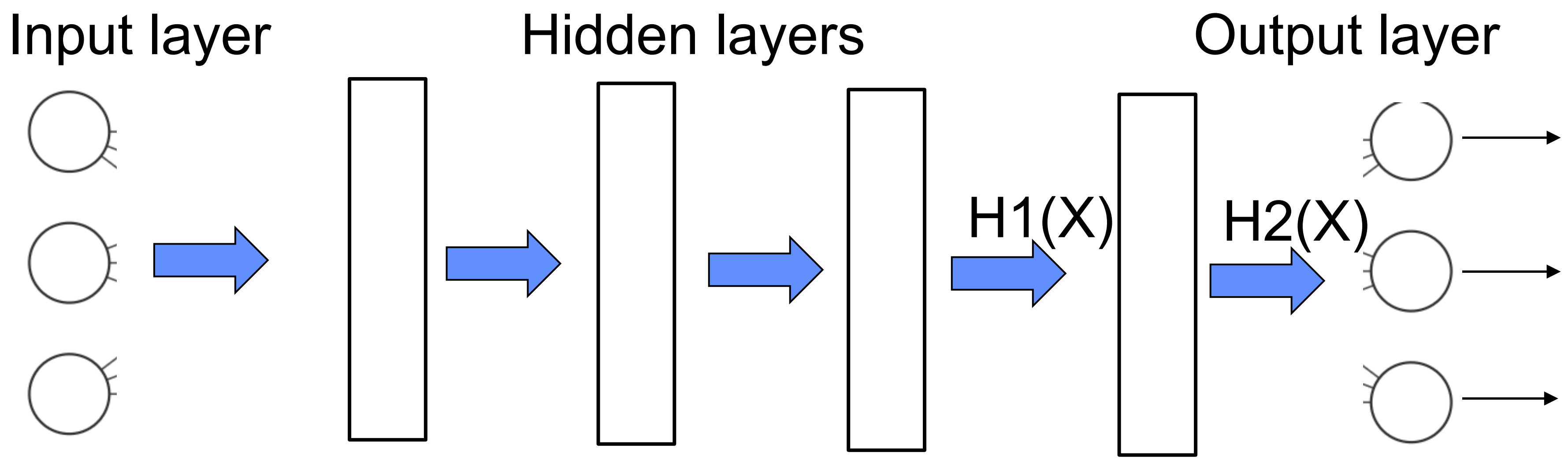
- **pause**

Consider: Can we keep adding deep layers?



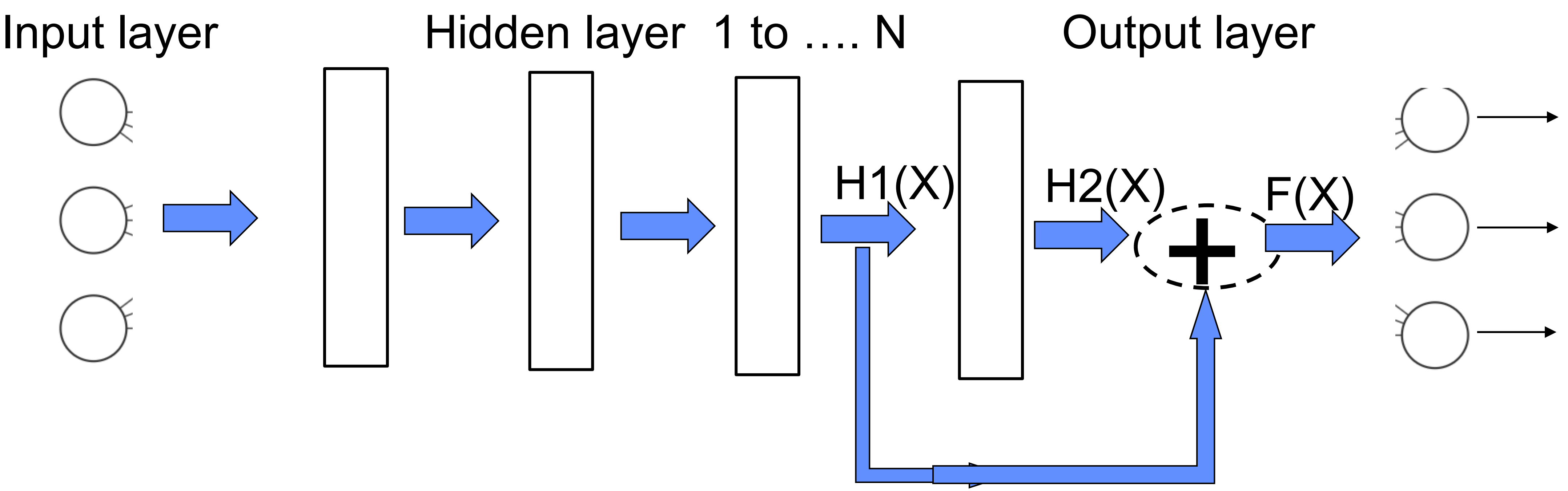
Given some deep network,
should I add another layer?
What should a new layer learn?

Consider: Can we keep adding deep layers?



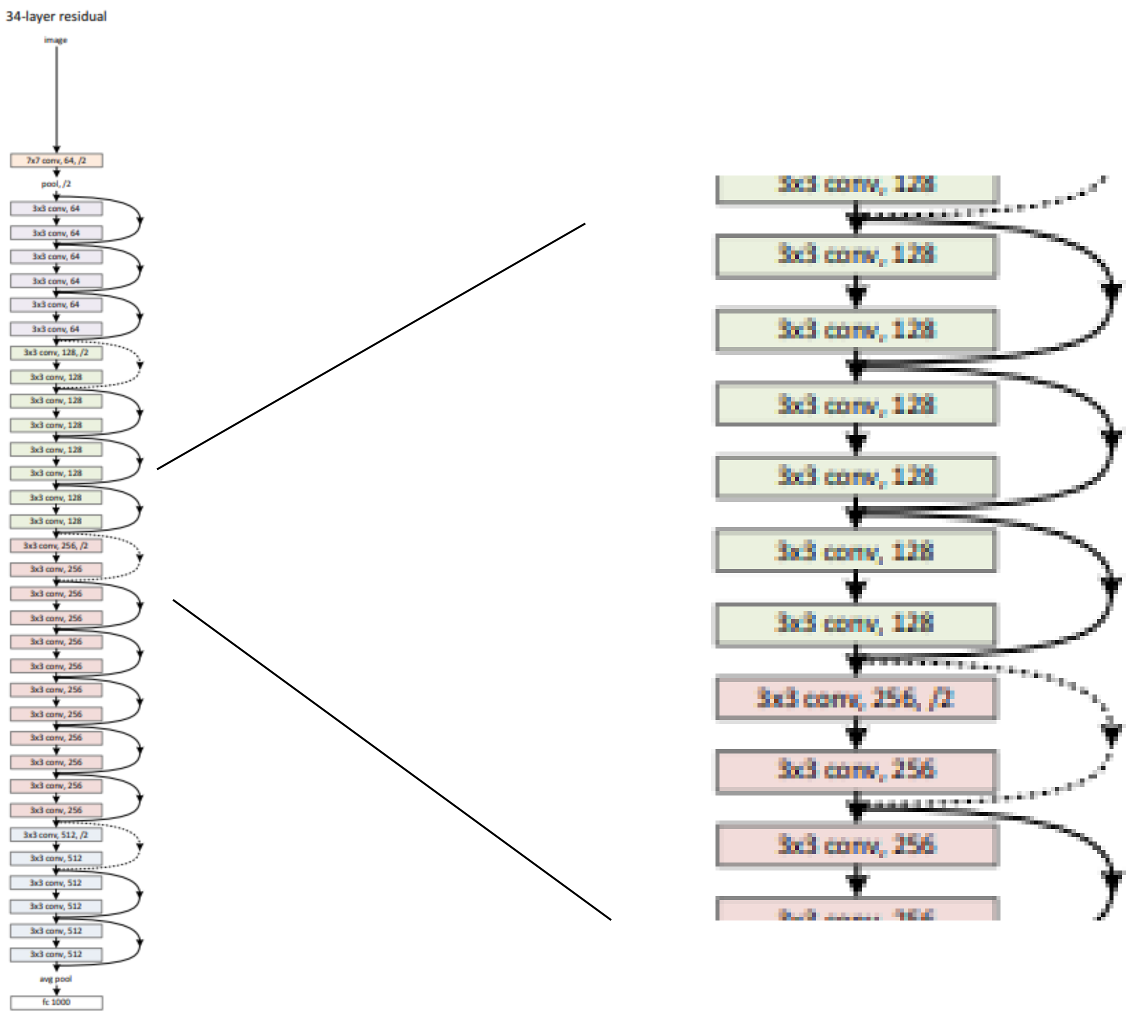
If H1(X) is good then this new layer could be unnecessary,
Eg H2(X) should be just H1(X)

Skip with addition makes a 'residual' connection



Make it easy for next layer to learn nothing –
e.g. use F(X)=H2(X)+H1(X) so that H2(X)=F(X)-H1(X) .
The H2() function learned is a residual function

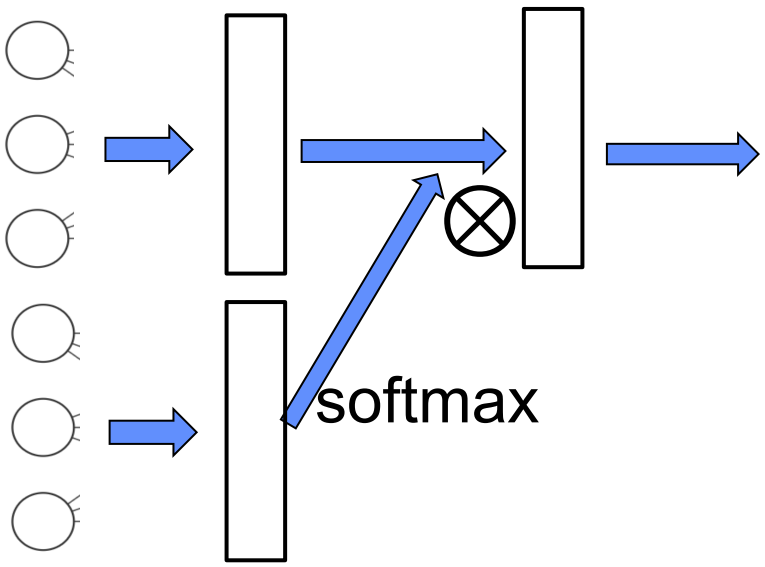
“Resnet” residual connections help deeper learning



Deep Residual Learning, He et.al, 2015

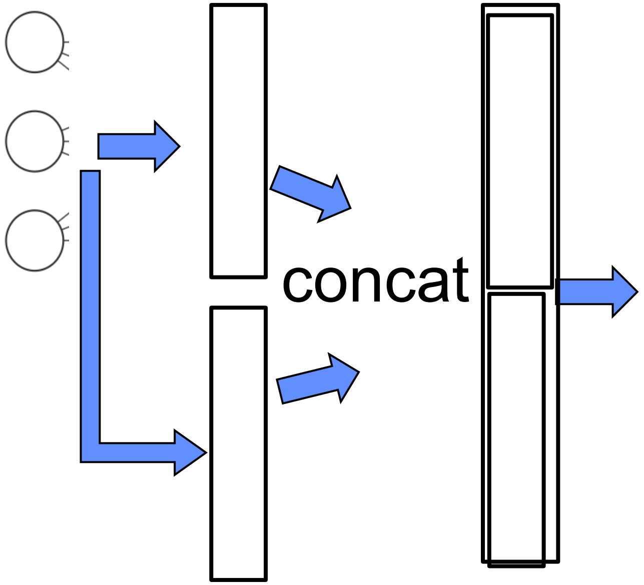
Summary: useful connections for architectures, and the intuitions

Softmax for gating



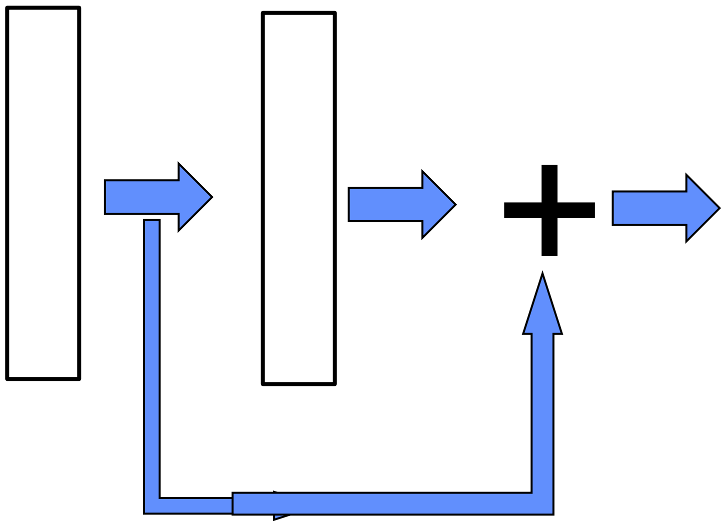
Recurrent nets,
language transformer nets

Skip connections for feature reuse



UNET, also
feedforward nets..

Residual connections help deeper learning



Resnet, large image
classification

- **Programing connections and Keras Model API**

Keras: Sequential API VS Functional API

```
#specify the neural network model and learning parameters
my_model  = tf.keras.models.Sequential([
                tf.keras.layers.Flatten(input_shape=(28, 28)),
                tf.keras.layers.Dense(32,activation='relu'),
                tf.keras.layers.Dense(10,activation='softmax')])
my_model.summary()
```

A sequence of layers: the inputs are assumed to be in order

Keras: Sequential API VS Functional API

```
#specify the neural network model and learning parameters
my_model  = tf.keras.models.Sequential([
                tf.keras.layers.Flatten(input_shape=(28, 28)),
                tf.keras.layers.Dense(32,activation='relu'),
                tf.keras.layers.Dense(10,activation='softmax')])
my_model.summary()
```

A sequence of layers: the inputs are assumed to be in order

```
#specify the neural network model and learning parameters
inputs             = tf.keras.layers.Input(shape=(28, 28, 1,))
inputs_flattened  = tf.keras.layers.Flatten()(inputs)
hidden_layer       = tf.keras.layers.Dense(32,activation='relu')(inputs_flattened)
output_layer       = tf.keras.layers.Dense(10,activation='softmax')(hidden_layer)
```

A sequence of functions: Input layer(s) are specified

Keras: Sequential API VS Functional API

```
#specify the neural network model and learning parameters
my_model   = tf.keras.models.Sequential([
                   tf.keras.layers.Flatten(input_shape=(28, 28)),
                   tf.keras.layers.Dense(32,activation='relu'),
                   tf.keras.layers.Dense(10,activation='softmax')])
my_model.summary()
```

A sequence of layers: the inputs are assumed to be in order

```
#specify the neural network model and learning parameters
inputs               = tf.keras.layers.Input(shape=(28, 28, 1,))
inputs_flattened    = tf.keras.layers.Flatten()(inputs)
hidden_layer         = tf.keras.layers.Dense(32,activation='relu')(inputs_flattened)
output_layer         = tf.keras.layers.Dense(10,activation='softmax')(hidden_layer)

my_model = tf.keras.Model(inputs,output_layer)
my_model.summary()
```

A sequence of functions: Input layer(s) are specified

The Model() function figures out the full path(s) to connect the input(s) to output(s)

Exercise

- MNIST autoencoder, reconstruct digits from noisy inputs

- Add skip connections with concatenation

Note: make sure you see how the outputs from encoding layers are matched up to inputs for decoding layers!

14x14 encoding feature maps should be concatenated with 14x14 decoding maps

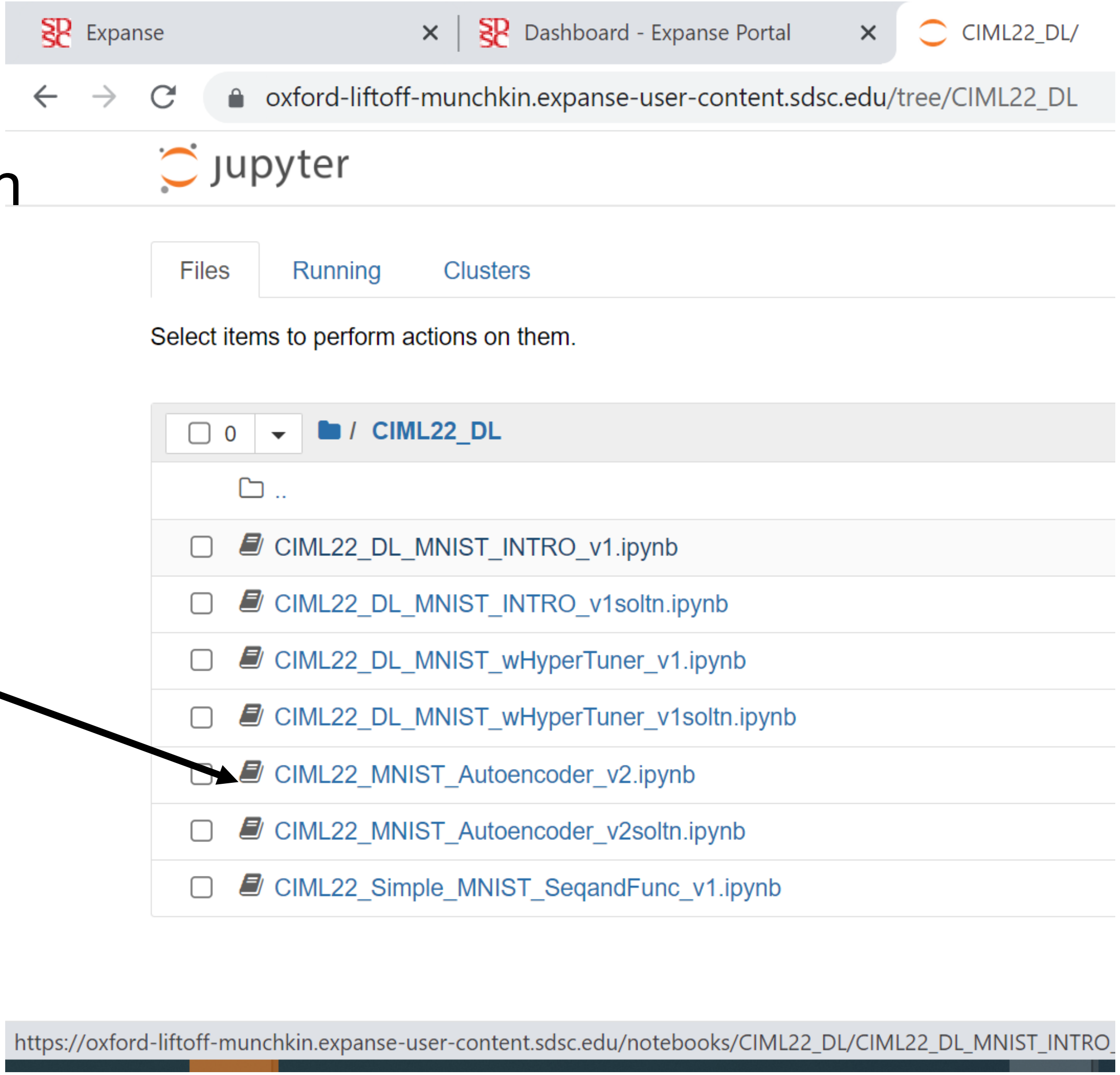
- Review outputs to see improvements

Login to expanse and start a notebook on gpu-shared queue

$ jupyter-gpu-shared-tensorflow

In jupyter notebook session open the MNIST_Autoencoder notebook

Follow instructions in the notebook



Quick overview of code

```
def encoder(inputs):
  '''Defines the encoder with two Conv2D and max pooling layers.'''
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding='same')(inputs)
                                                    #padding same produces same output size
```

Encoder function

with convolutions etc…

```
def decoder(inputs, enc_conv1,enc_conv2):
  '''Defines the decoder path to upsample back to the original image size.'''
  #Notice that padding = same keeps the output same size as input

  conv_1      = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu', padding='
  up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)
```

Decoder function

with up sampling (deconvolutions) etc…

```python
def encoder(inputs):
  '''Defines the encoder with two Conv2D and max pooling layers.'''
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding='same')(inputs)
                                                                   #padding same produces same output size
  max_pool_1 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_1) #max pooling does the downsampling

  conv_2 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu', padding='same')(max_pool_1)
  max_pool_2 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_2)

  return max_pool_2, conv_1, conv_2
```

Return final output and intermediate layer outputs,

```python
def decoder(inputs, enc_conv1,enc_conv2):
    '''Defines the decoder path to upsample back to the original image size.'''
    #Notice that padding = same keeps the output same size as input

    conv_1      = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu', padding='
    up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)
```

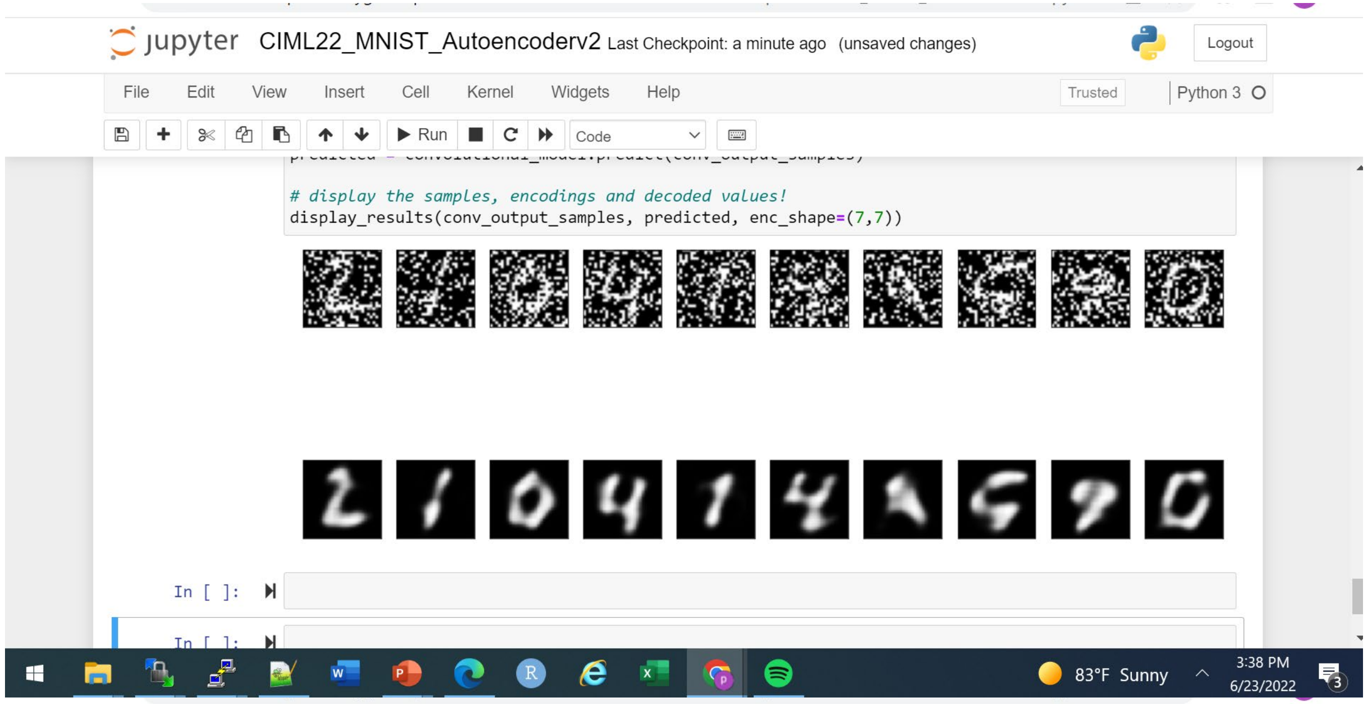
```python
def encoder(inputs):
  '''Defines the encoder with two Conv2D and max pooling layers.'''
  conv_1 = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding='same')(inputs)
                                                                  #padding same produces same output size
  max_pool_1 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_1) #max pooling does the downsampling

  conv_2 = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu', padding='same')(max_pool_1)
  max_pool_2 = tf.keras.layers.MaxPooling2D(pool_size=(2,2))(conv_2)

  return max_pool_2, conv_1, conv_2
```

Return final output and intermediate layer outputs,

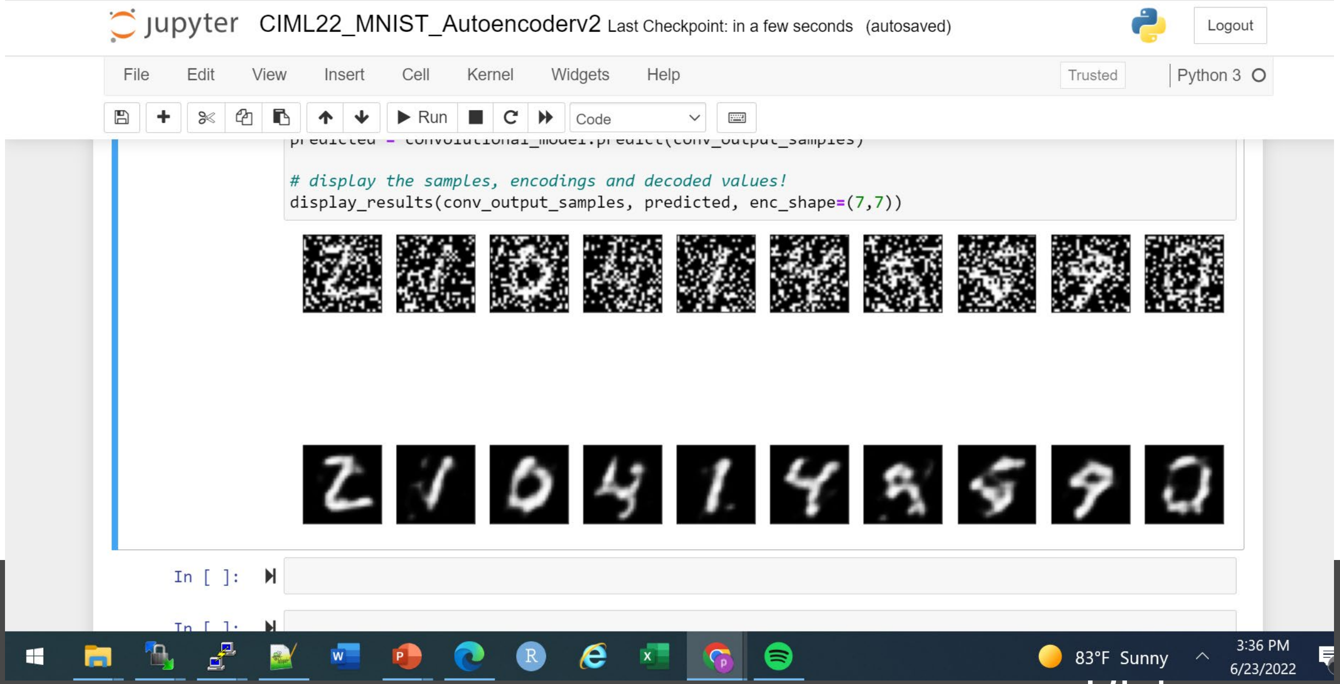
```python
def decoder(inputs, enc_conv1,enc_conv2):
  '''Defines the decoder path to upsample back to the original image size.'''
  #Notice that padding = same keeps the output same size as input

  conv_1      = tf.keras.layers.Conv2D(filters=128, kernel_size=(3,3), activation='relu', padding='
  up_sample_1 = tf.keras.layers.UpSampling2D(size=(2,2))(conv_1)

#another optoin is transpose
# up_sample_1 = tf.keras.layers.Conv2DTranspose(128,kernel_size=(2,2),strides=(2,2))(conv_1)
# in a transpose convolutional layer,

  # --------->>>>> before the conv_2 line add a
  # --------->>>>          tf.keras.layers.concatenate statement to combine enc_conv2 with decoding u
  #  conv_2      = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding

  skip_concat_1 = tf.keras.layers.concatenate([up_sample_1, enc_conv2])
  conv_2      = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu', padding='s
                                        # ---------->>>> and change the input into conv_2
```

and pass intermediate layers on

then use it in concatenation layer



With out skip
20 epochs
Loss 0.1664



With skip,
20 epochs loss 0.14

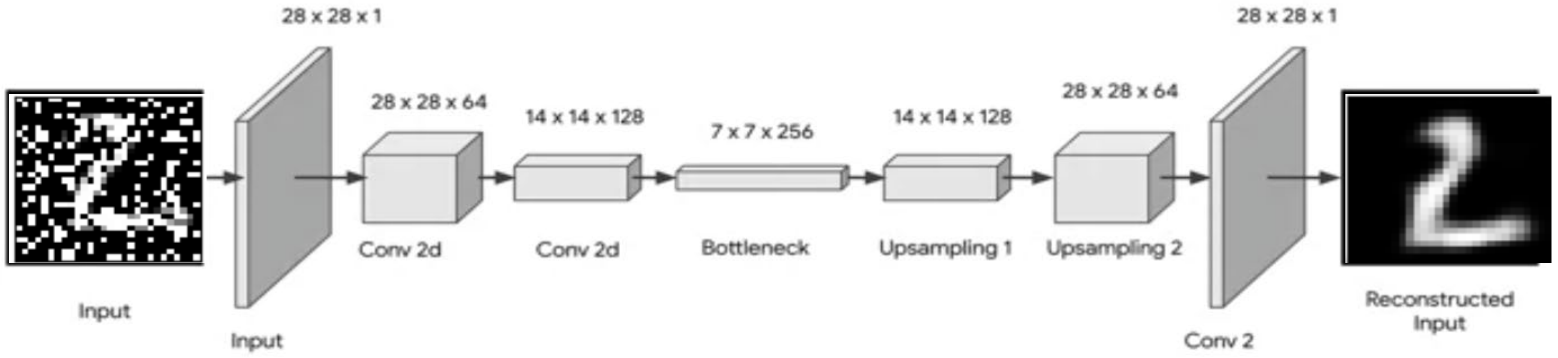
Are the numbers a little bit more reconstructed?

Autoencoding with Stable Diffusion

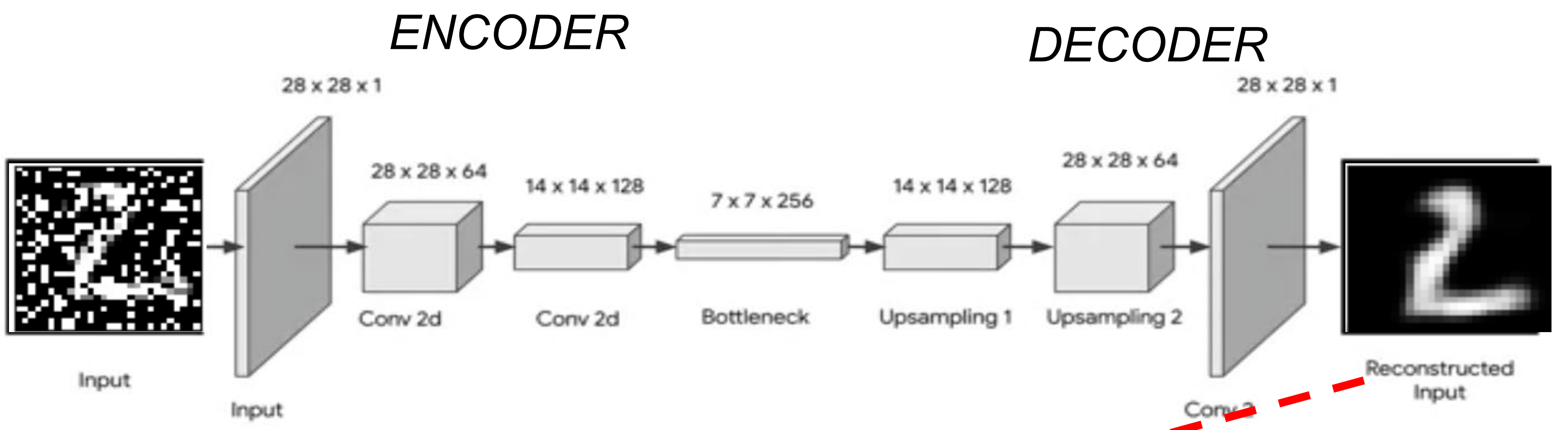
- Let’s introduce the concepts and intuition behind stable diffusion

In principle, our denoising autoencoder removed noise pixels and/or filled in number pixels

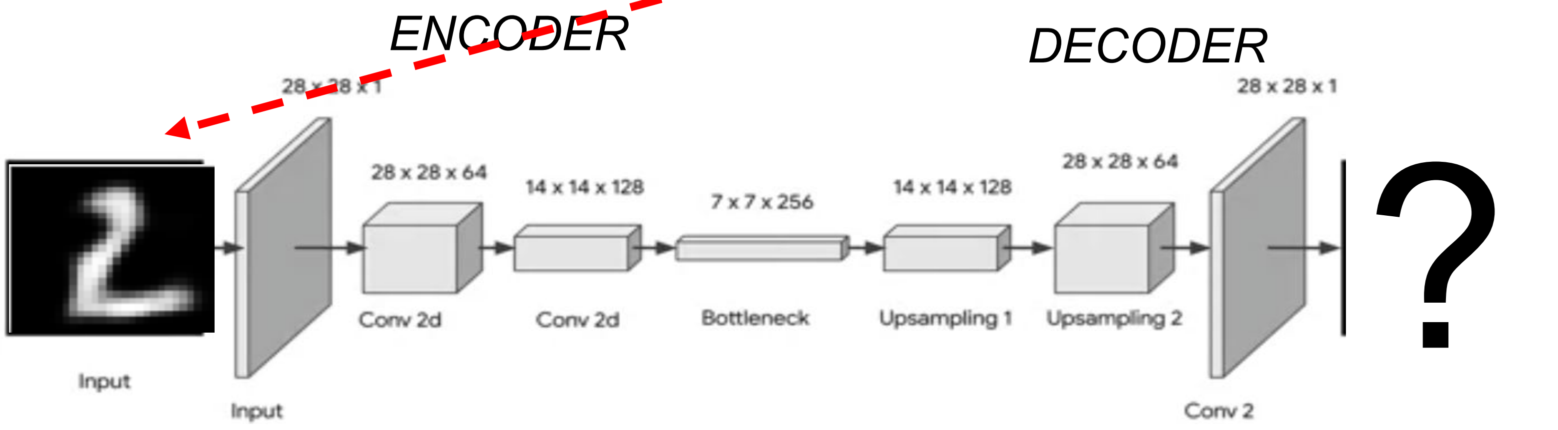
ENCODER *DECODER*



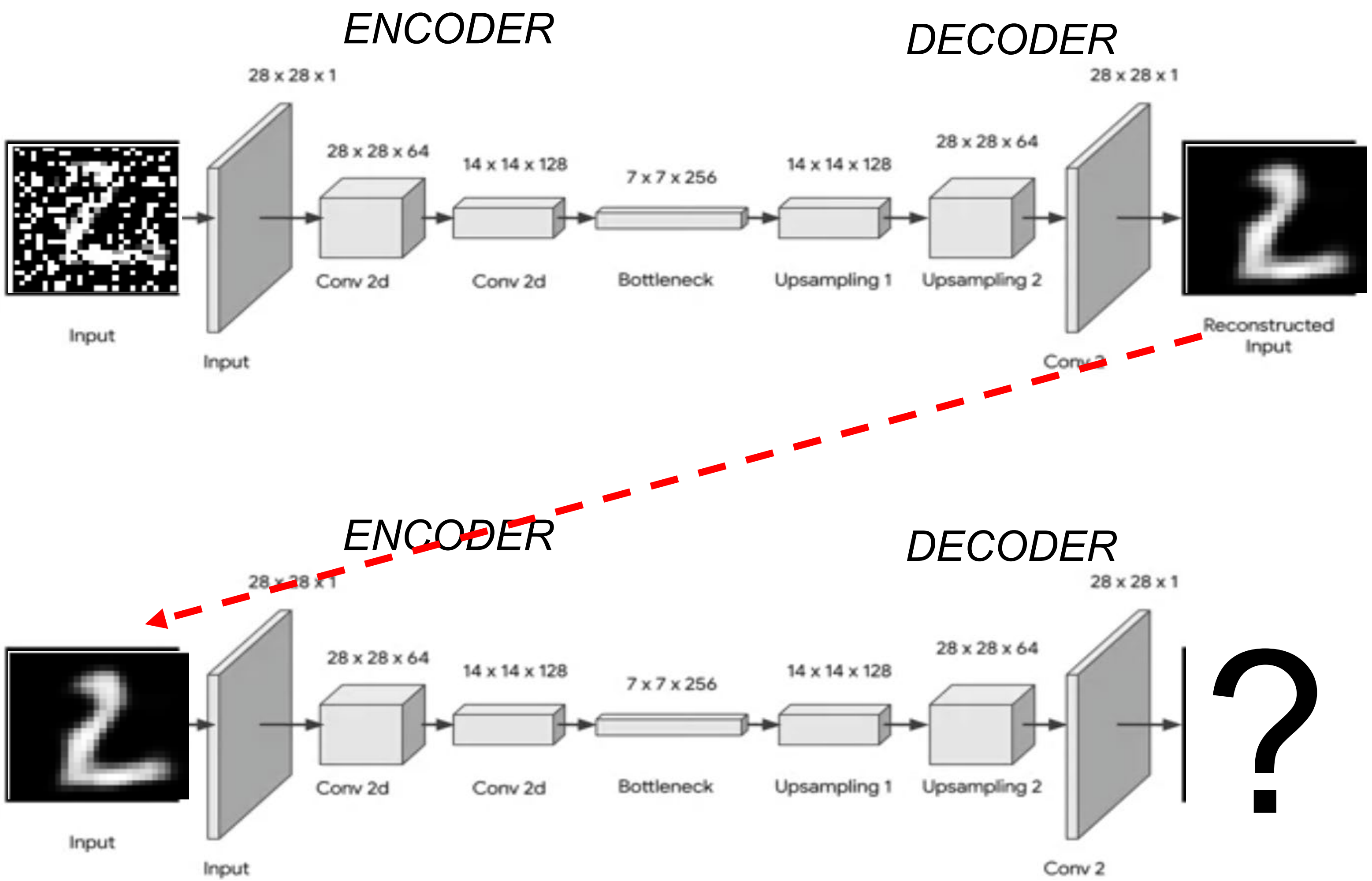
In principle, our denoising autoencoder removed noise pixels and/or filled in number pixels



What would would happen if we fed the denoised output back into the autoencoder?



In principle, our denoising autoencoder removed noise pixels and/or filled in number pixels



What would would happen if we fed the denoised output back into the autoencoder?

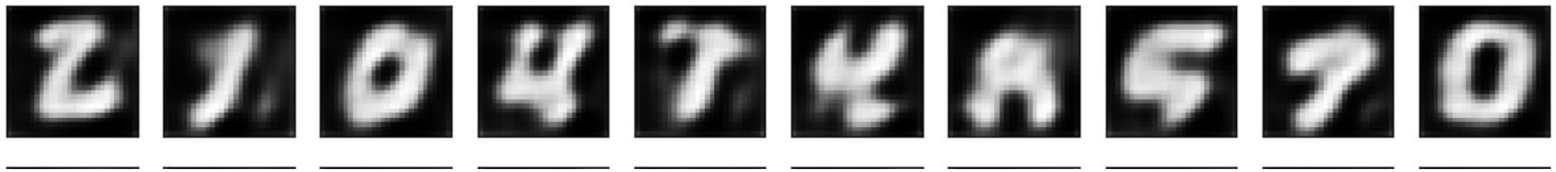
A: better reconstruction

B: all pixels would be removed

C: all pixels would be filled in

First step of denoising

first step of denoising

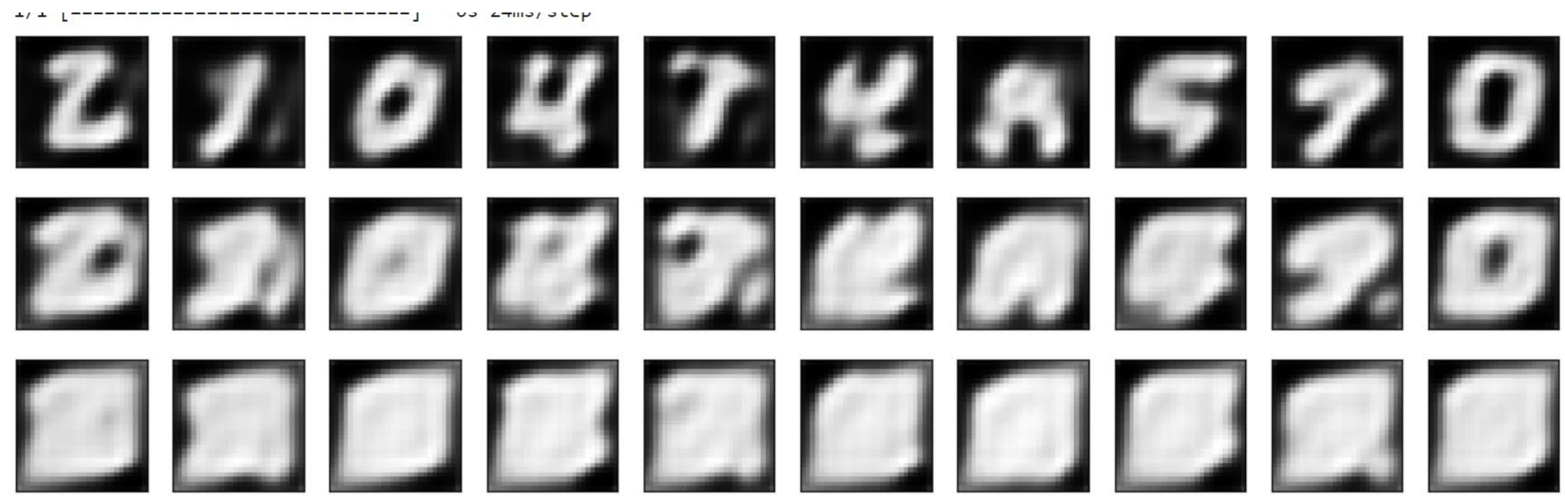


1 more step of denoising

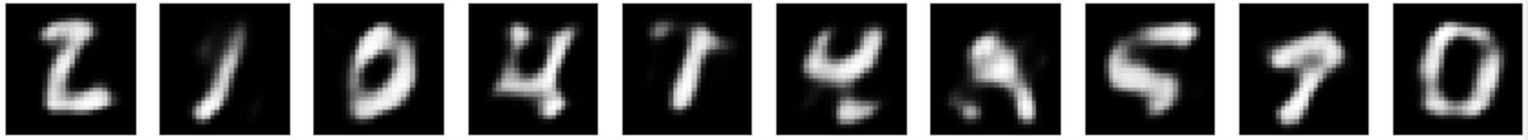
Is it better?

First step of denoising

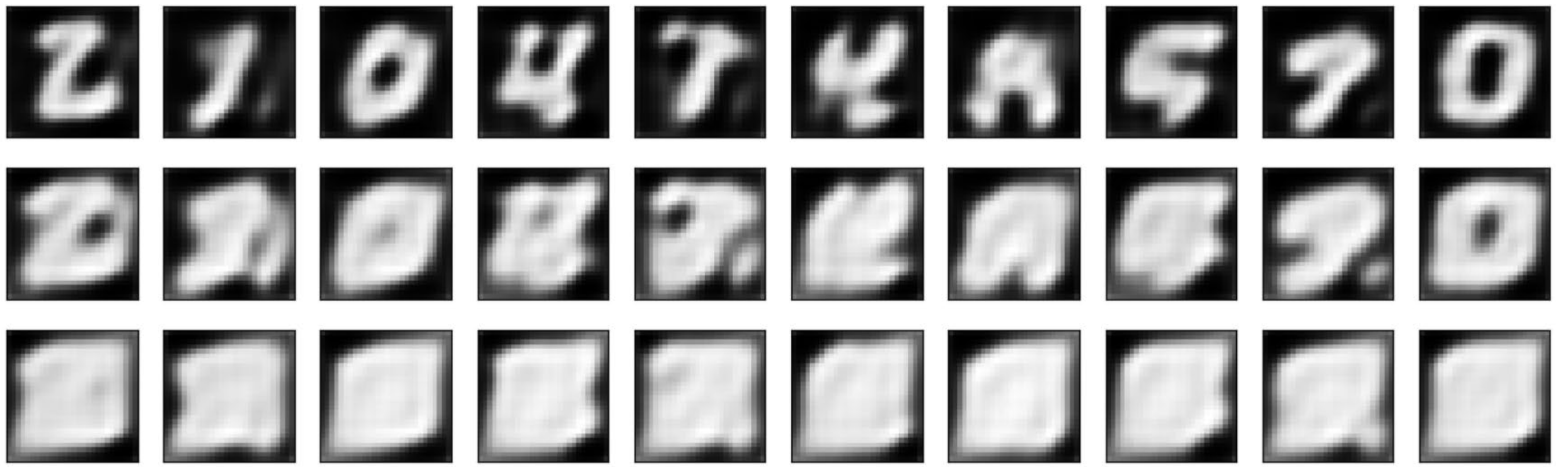
3 more steps of denoising



Frist step of denoising



3 more steps of denoising



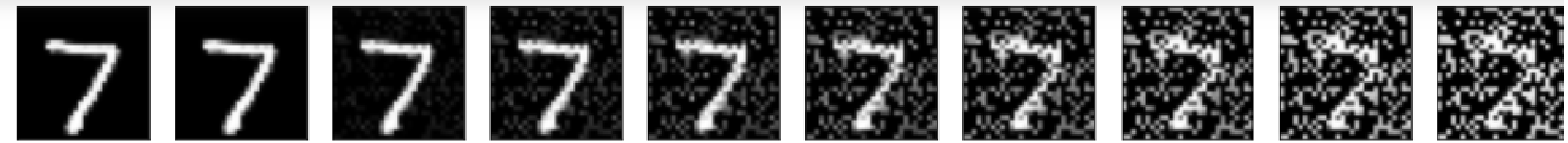
Let’s make this more stable, by training a network to just remove a little noise. It is like training to predict noise diffusion.

Stable Diffusion for Image Reconstruction

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. 2020.

- **Concept:**

create a sequence of images with noise, t=1…T



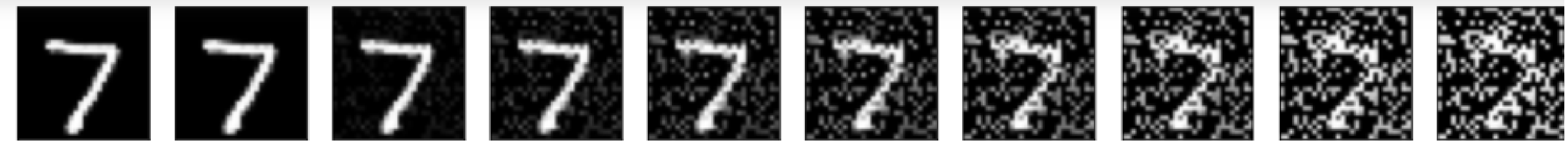
t=1 → t=T

Stable Diffusion for Image Reconstruction

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. 2020.

- **Concept:**

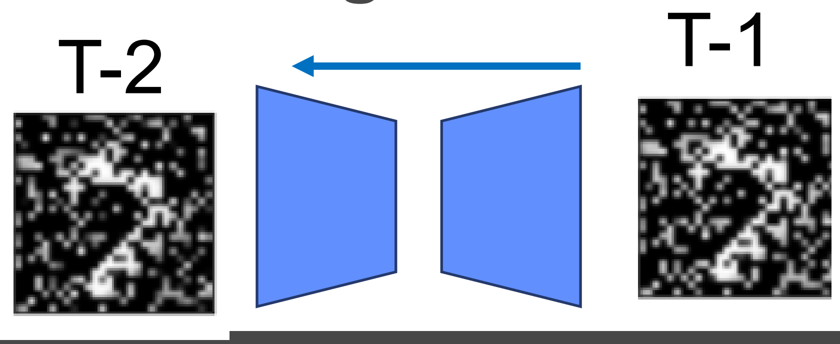
create a sequence of images with noise, t=1…T



t=1 → t=T

train the network to reconstruct image t-1 from image t

etc…



Note: this example is in pixel space, but it is often applied in embedding space

- From Ho et al. 2020

Early denoising steps add overall structure
Later denoising steps add more detail

Exercise

- **MNIST stable diffusion, or incremental denoising**
- **Open and run the notebook**
- **Try changing T parameter (steps2use)**

jupyter C24_AE_diff_v8 Last Checkpoint: 18 seconds ago

File Edit View Run Kernel Settings Help

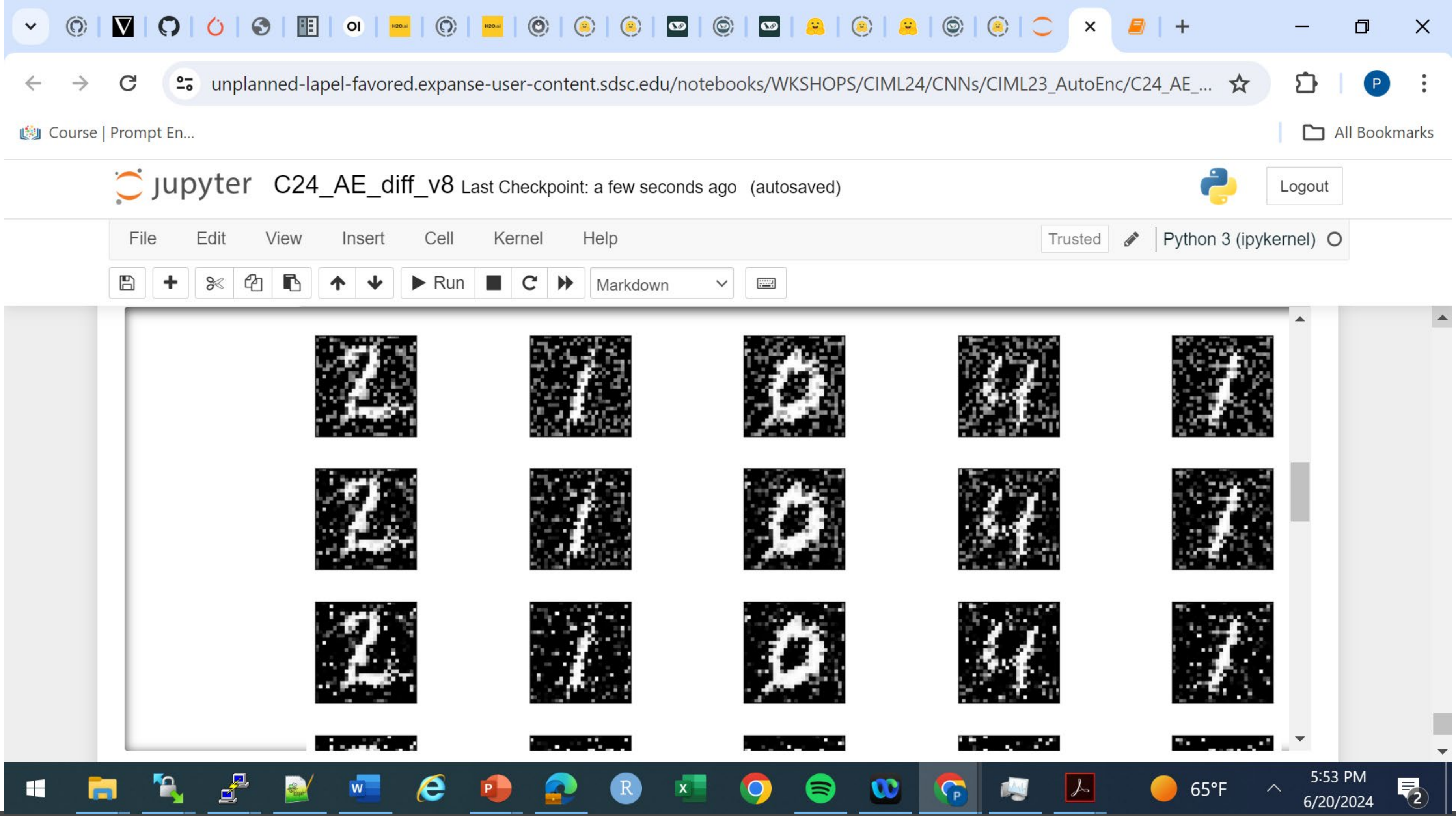
Code JupyterLab Python 3 (ipyk

Set up the diffusion/incremental noise here. The steps2use is the T parameter

[5]:
```
# Either set up diffusion images or just addnoise
if 1:
  dist2try      ='norm'  # use normal distribution for noise
  mag2use       = 1;     #magnitude of noise, aka scale or standard deviation
  steps2use     = 10;    #number of diffusion steps to mimic   <<<<<--------------------

  newN = X_train.shape[0]*(steps2use+1)
```

Sample output where T-1,T-2, are going down the columns

What would happen if the input was completely random?



- end