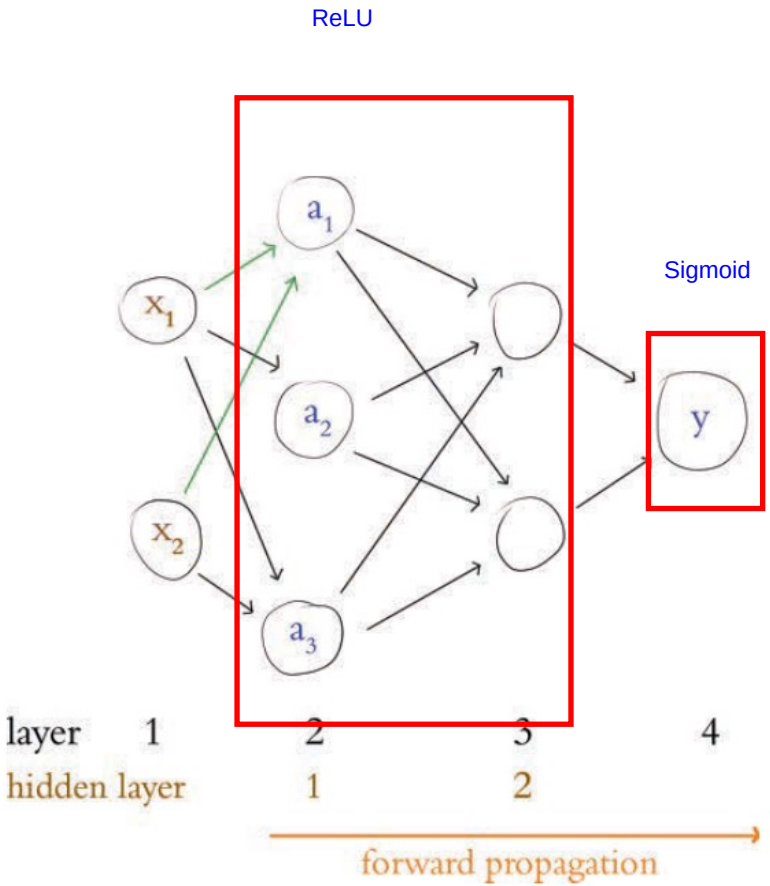
MATH 4280

Lecture Notes 7: More neural network concepts

Dense network

- There are two input neurons x_1, x_2
- There are two dense hidden layers
- The first hidden layer has 3 ReLU neurons
- The second hidden layer has 2 ReLU neurons
- There is one output neuron y
- For binary classification, the output neuron should be sigmoid



Forward propagation

- Take the neuron a_1 as an example
- It takes two inputs x_1 and x_2
- This neuron computes

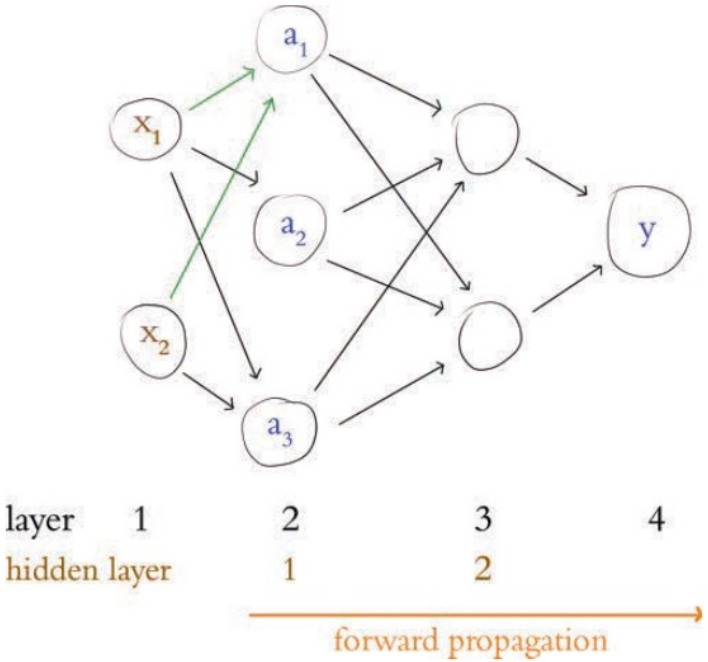
$$z = w \cdot x + b$$

$$z = (w_1 x_1 + w_2 x_2) + b$$

- Here w_1, w_2, b are weights
- Then the activation is applied

$$a = max(0, z)$$

- Similar computations are done for other neurons in hidden layers



Sigmoid output neuron

- Again, this neuron y computes

$$z = w \cdot x + b$$

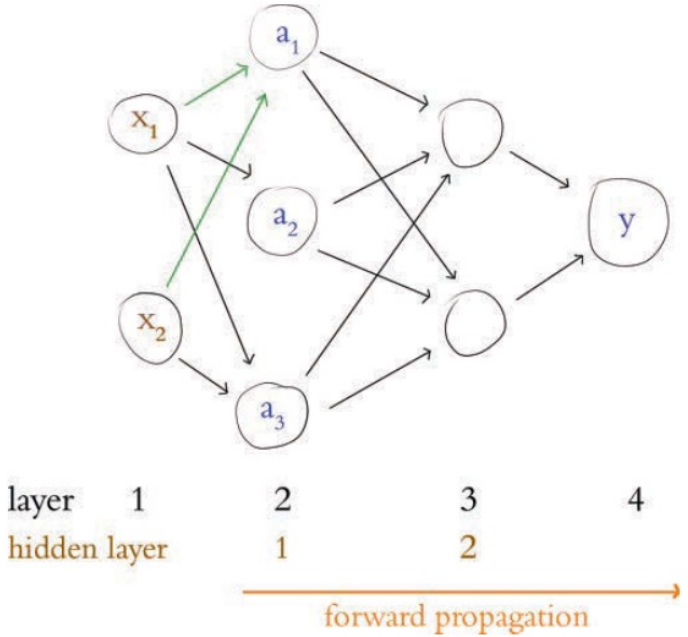
$$z = (w_1 x_1 + w_2 x_2) + b$$

- Then, the sigmoid function is applied

$$a = \sigma(z)$$

$$= \frac{1}{1 + e^{-z}}$$

- The output is used for binary classifications

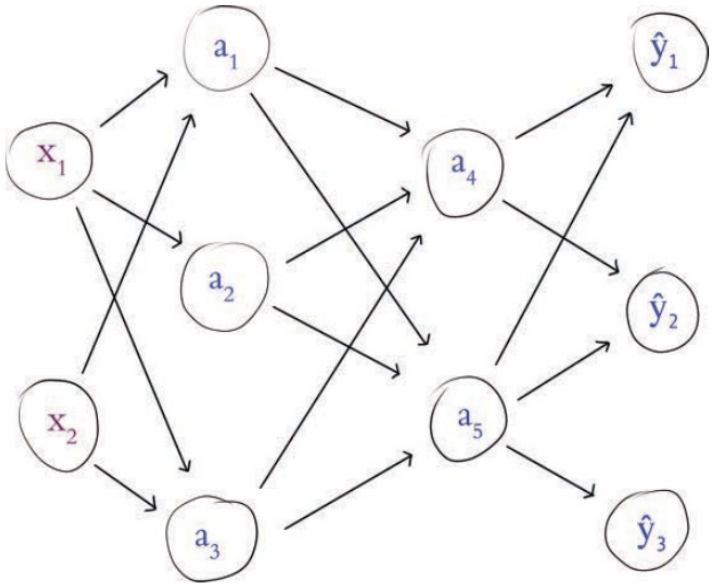


Softmax output neuron

- It is used for multi-class classification
- Compute $w \cdot x + b$ in each of the 3 output neurons
- Then we compute

$$y_i = e^{z_i} / (e^{z_1} + e^{z_2} + e^{z_3})$$

- These output values are used for classification



Trainable parameters

- Use summary() method

```
model = Sequential()
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

- In the first Dense layer, number of parameters is

$$784 \times 64 + 64 = 50240$$

- In the second Dense layer, number of parameters is

$$64 \times 10 + 10 = 650$$

Loss functions

- Quadratic cost (i.e. MSE)

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Here y_i is the true value and $\hat{y}_i$ is the network approximated value
- Cross-Entropy cost (e.g. used in sigmoid output neuron)

Idea: Try to match the predicted value to either 0 or 1

$$C = -\frac{1}{n}\sum_{i=1}^{n}[y_i \ln \hat{y}_i + (1 - y_i)\ln(1 - \hat{y}_i)]$$

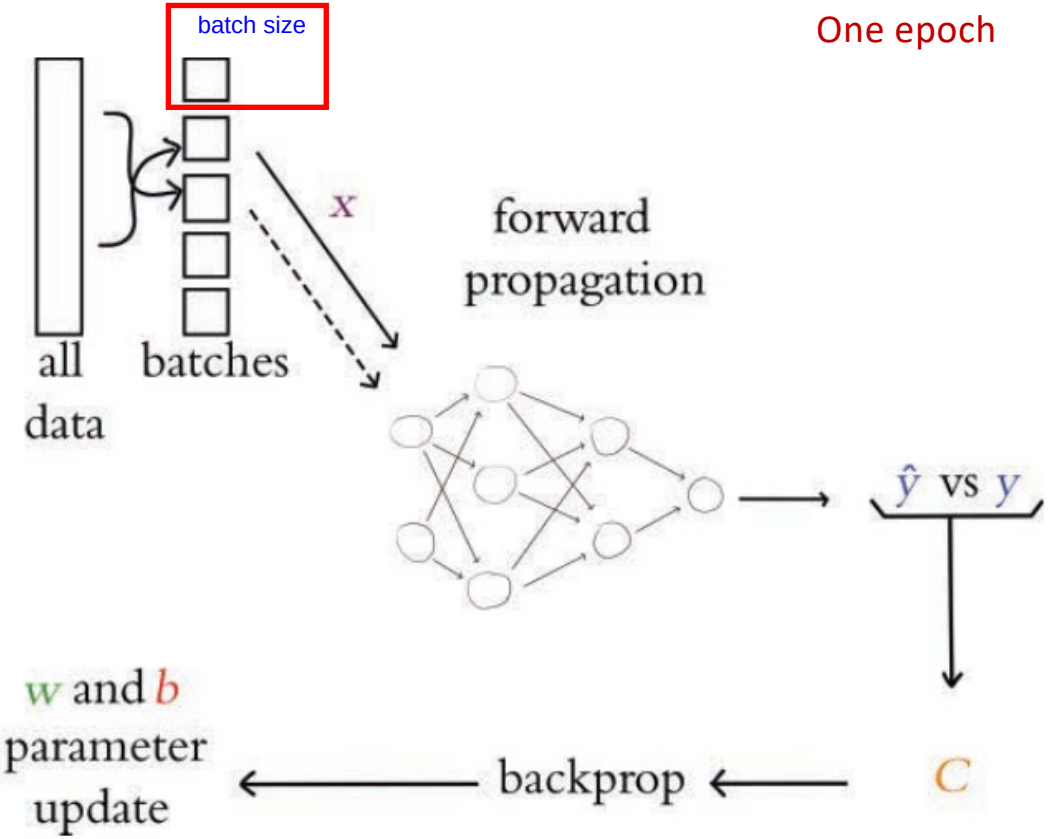
- The derivative is given below, giving faster convergence ????

$$\frac{dC}{d\hat{y}_i} = \frac{y_i - \hat{y}_i}{\hat{y}_i(1 - \hat{y}_i)}$$

Batch size and SGD

Round of Training:

1. Sample a mini-batch of x values
2. Forward propagate x through network to estimate y with $\hat{y}$
3. Calculate cost C by comparing y and $\hat{y}$
4. Descend gradient of C to adjust w and b, enabling x to better predict y



An example

- We first define the following model

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

- We can use model.summary() to see the model details

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_3 (Dense)              (None, 10)                650
=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
```

- Next, we can compile the model

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

- The loss function is selected as cross-entropy
- Use the stochastic gradient descent, the learning rate is 0.1
- Also include the model accuracy in addition to loss

- Next we perform the training

```
model.fit(X_train, y_train,
          batch_size=128, epochs=20,
          verbose=1,
          validation_data=(X_valid, y_valid))
```

number of iterations = 20

show the training information at each step

- We can read the performance of the training

```
Epoch 1/20
60000/60000 [==============================] - 1s 15us/step - loss: 0.4744 - acc: 0.8637 - val_loss: 0.2686 - val_acc: 0.9234
Epoch 2/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.2414 - acc: 0.9289 - val_loss: 0.2004 - val_acc: 0.9404
Epoch 3/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1871 - acc: 0.9452 - val_loss: 0.1578 - val_acc: 0.9521
Epoch 4/20
60000/60000 [==============================] - 1s 12us/step - loss: 0.1538 - acc: 0.9551 - val_loss: 0.1435 - val_acc: 0.9574
```

Weight initialization

- In training, the parameters w and b are initialized with random values
- Import the packages

```
import numpy as np
import matplotlib.pyplot as plt
from keras import Sequential
from keras.layers import Dense, Activation
from keras.initializers import Zeros, RandomNormal
```

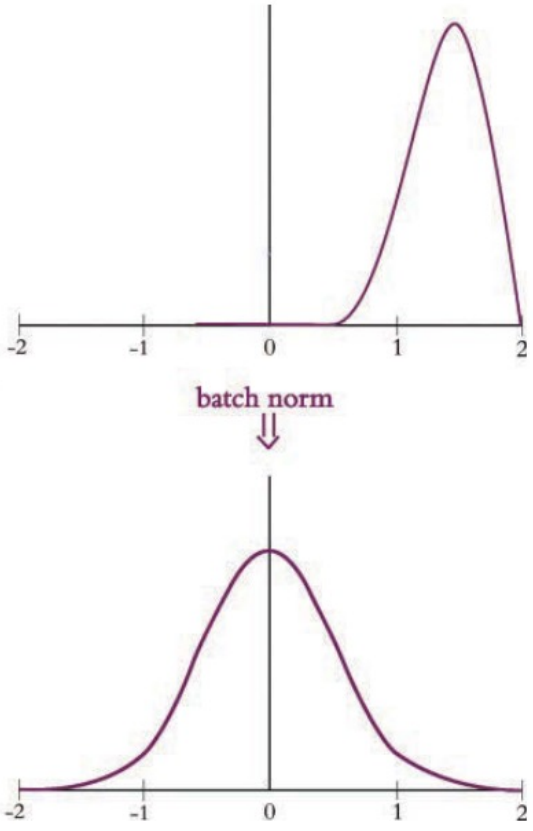
- Then we can do the initialization as follows

```
b_init = Zeros()
w_init = RandomNormal(stddev=1.0)
```

```
model = Sequential()
model.add(Dense(n_dense,
                input_dim=n_input,
                kernel_initializer=w_init,
                bias_initializer=b_init))
model.add(Activation('sigmoid'))
```

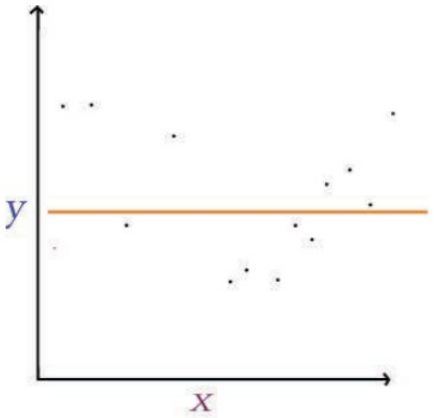
Batch normalization

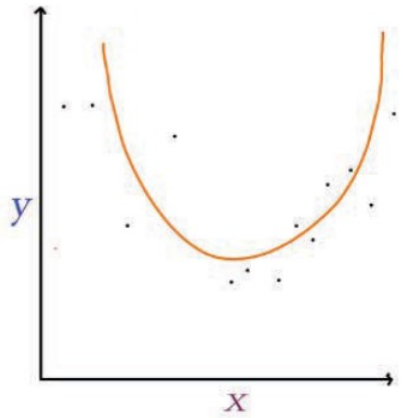
- Batch norm takes the activations output from the preceding layer, substracts the batch mean, and divides by the batch standard derivation
- Recenter the distribution with mean 0 and standard derivation 1
- Large values in one layer won't excessively influence the calculations in next layer
- This allows for selection of a higher learning rate
- The batch norm learns two more parameters γ and β, which play the role of standard derivation and mean



Model generalization

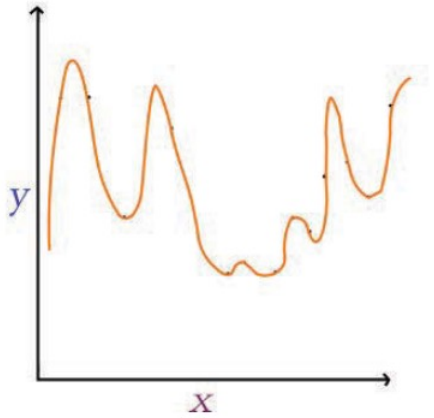
Model with two few parameters

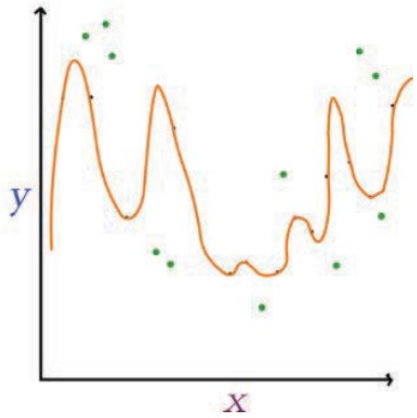




Model fits well

Overfitted model



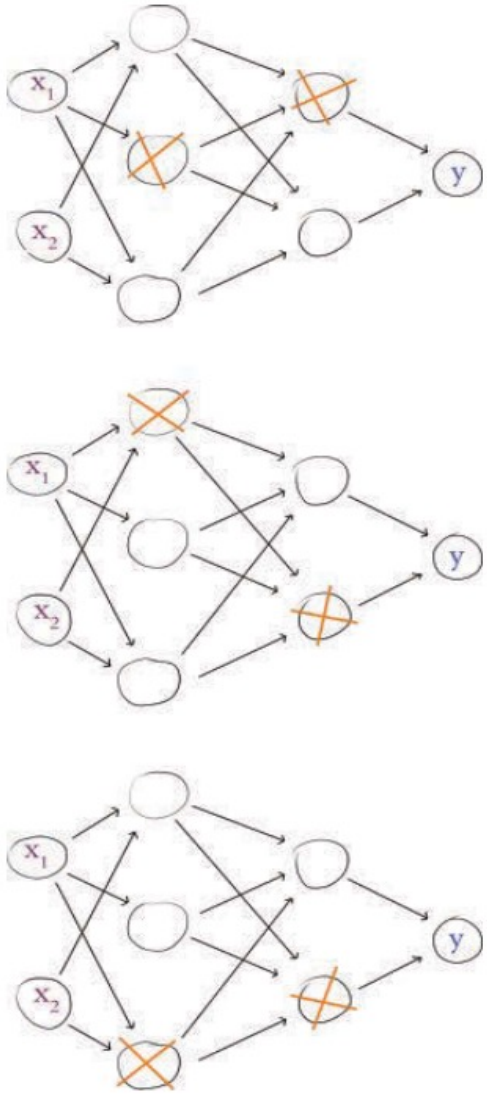


Overfitted model generalizes poorly to new data points

Avoiding overfitting

- Using L1 minimization
- Using Dropout: simply drop randomly selected neurons
- No need to apply dropout to all layers
- Usually, applied to later layers
- Usually, 20% - 50% dropout

avoid overfitting



Some optimizers

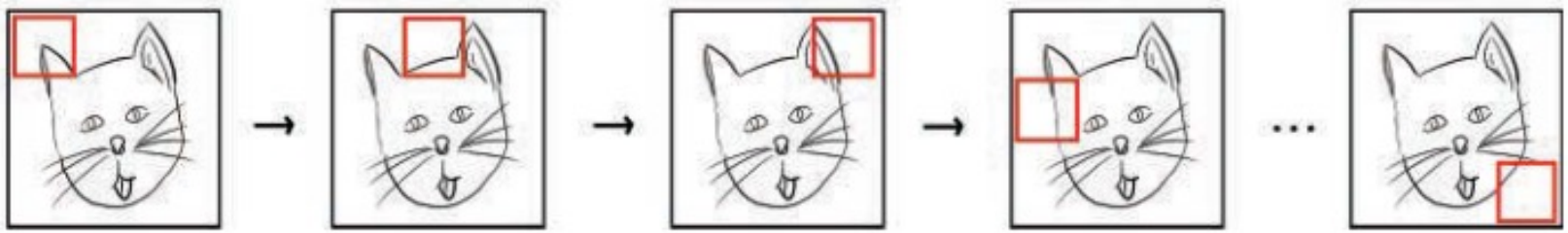
- Momentum: using moving average of the gradients
- There is an additional hyperparameter β, between 0 and 1
- Smaller β permits older gradients to contribute to the average
- AdaGrad = adaptive gradient, every parameter has a unique learning rate
- AdaDelta and RMSProp combine AdaGrad and moving average
- Adam is similar to AdaDelta and RMSProp, with two exceptions:
- 1. An extra moving average for each parameter
- 2. A clever bias trick to prevent moving averages from skewing toward zero at the start of training

An example

```
from keras.layers import Dropout
from keras.layers.normalization import BatchNormalization

model = Sequential()

model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(BatchNormalization())

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))
```

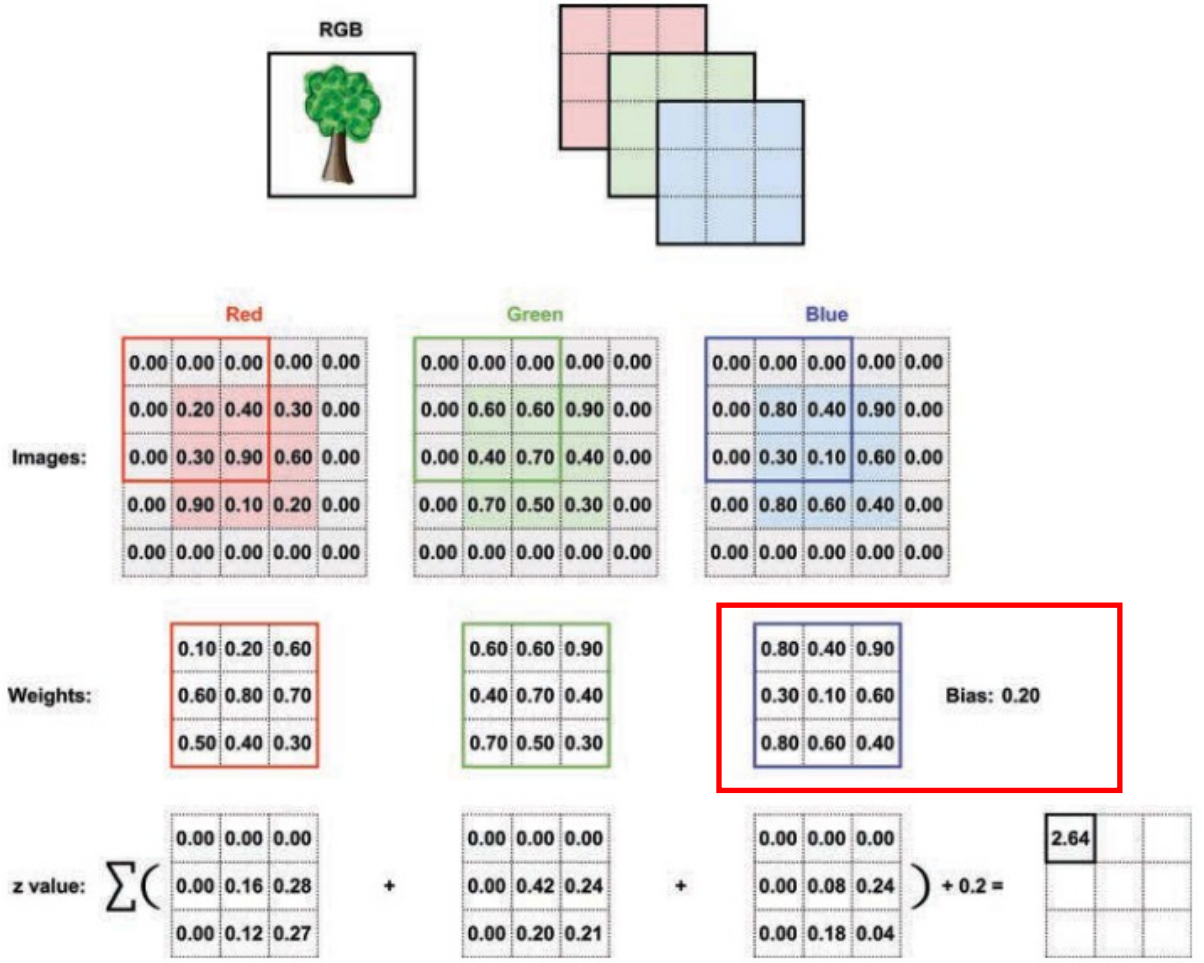
Convolutional neural networks

- CNN contains one or more convolutional layers
- It allows processing of spatial patterns
- Convolutional layers consist of sets of kernels (or filters)
- Each of these is a small window (or patch) that scans across the image
- Typically, there are multiple kernels in a given convolutional layer

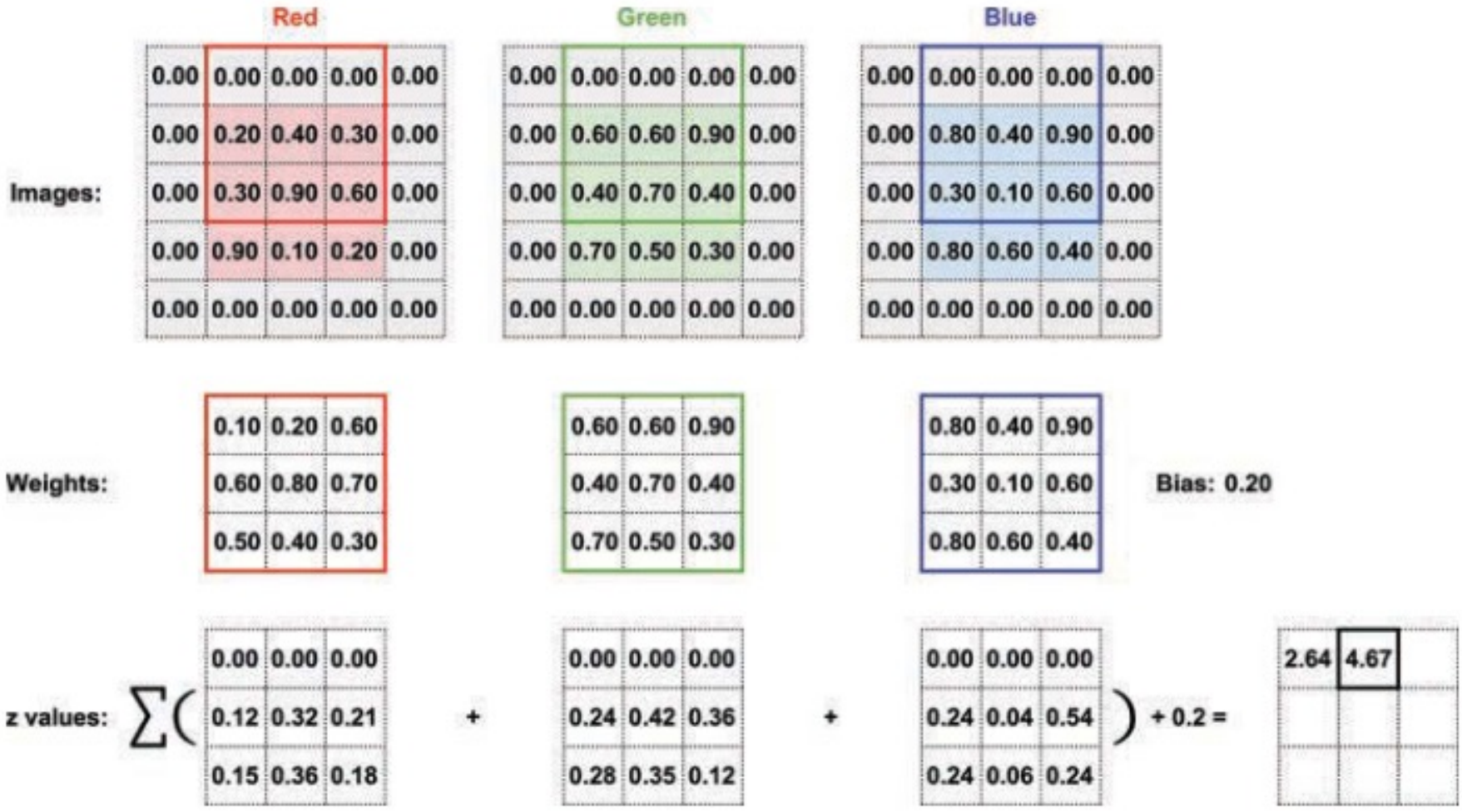


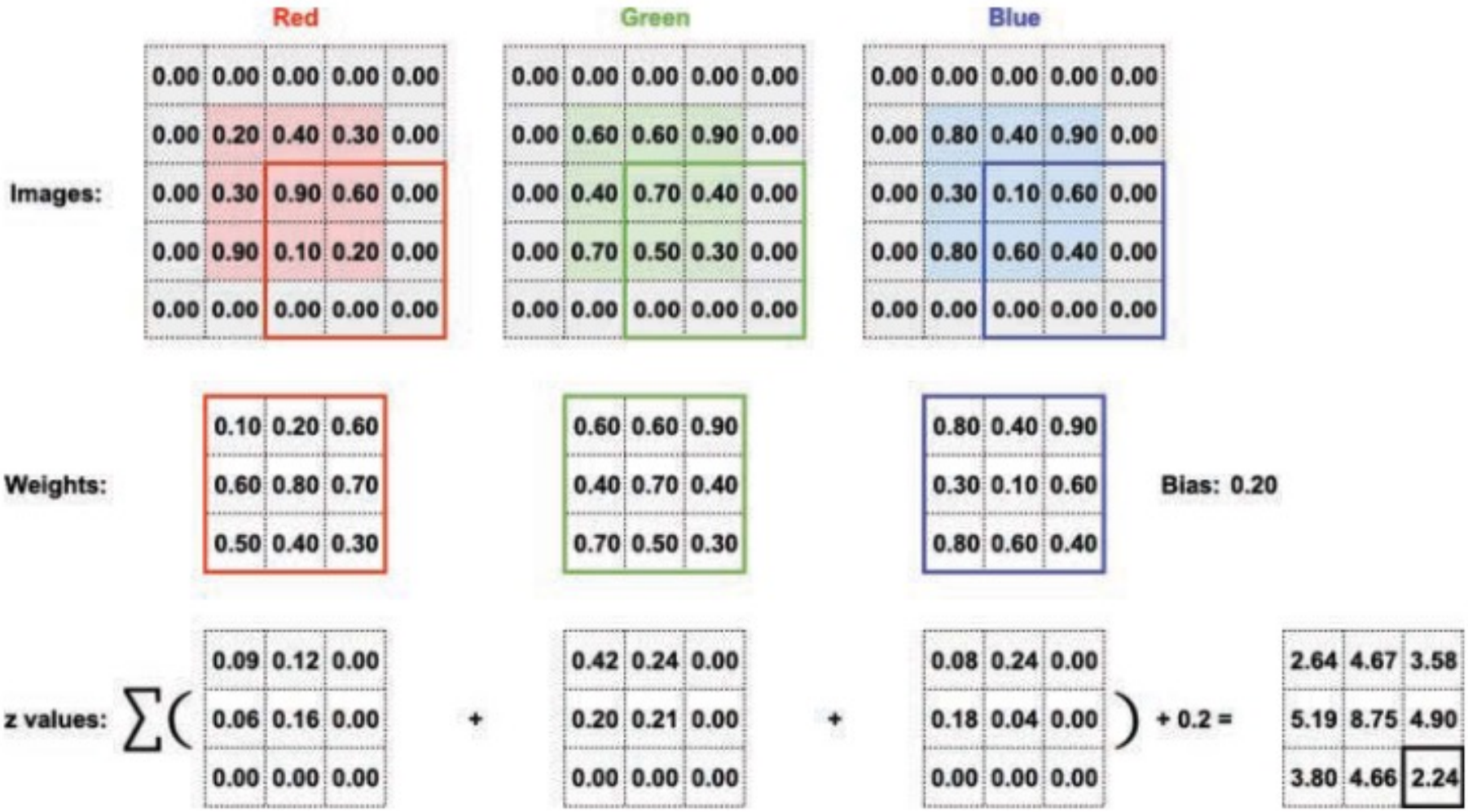
A convolutional example

- Take a color image
- Apply a 3x3 filter to each color

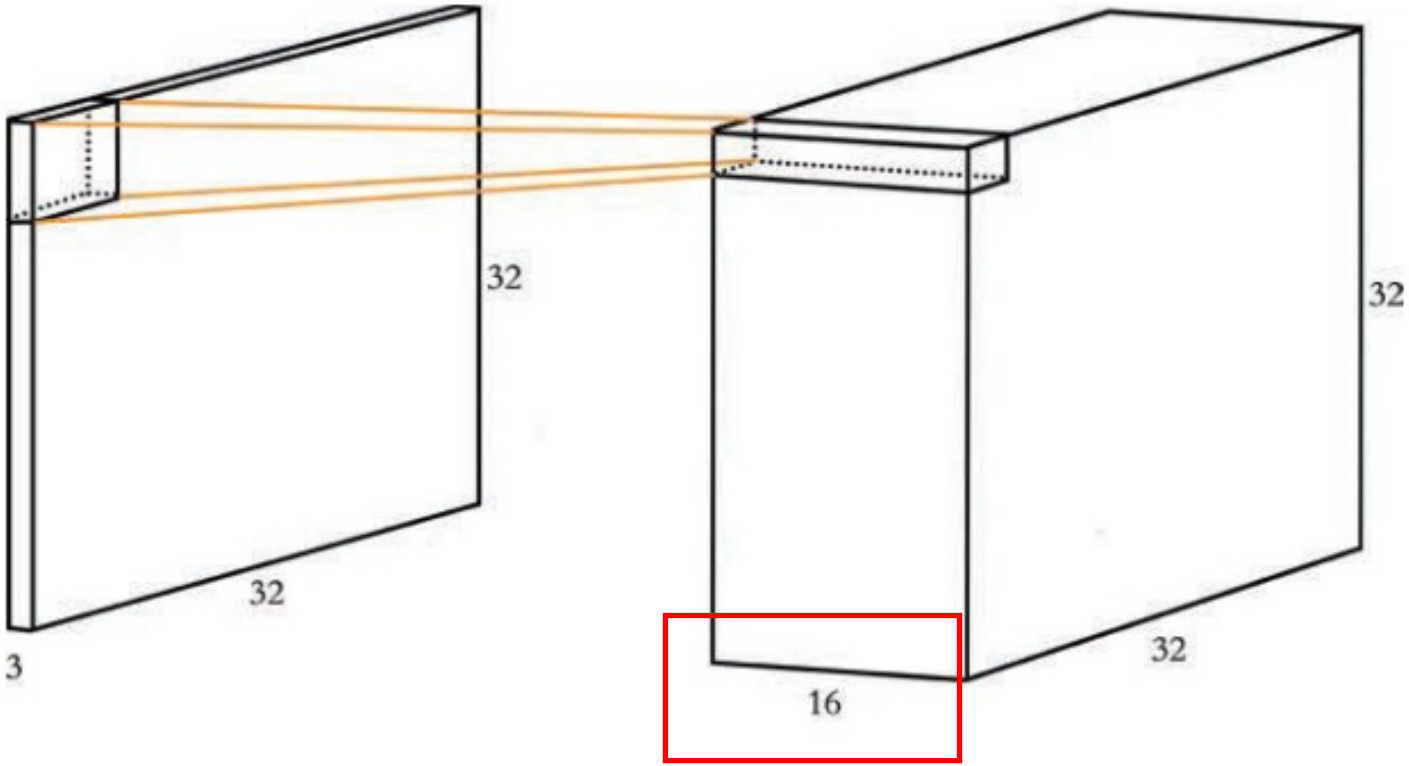


Each layer can be considered as a transformation


Red
Green
Blue
Images:
0.00 0.00 0.00 0.00 0.00
0.00 0.20 0.40 0.30 0.00
0.00 0.30 0.90 0.60 0.00
0.00 0.90 0.10 0.20 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.60 0.60 0.90 0.00
0.00 0.40 0.70 0.40 0.00
0.00 0.70 0.50 0.30 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.80 0.40 0.90 0.00
0.00 0.30 0.10 0.60 0.00
0.00 0.80 0.60 0.40 0.00
0.00 0.00 0.00 0.00 0.00
Weights:
0.10 0.20 0.60
0.60 0.80 0.70
0.50 0.40 0.30
0.60 0.60 0.90
0.40 0.70 0.40
0.70 0.50 0.30
0.80 0.40 0.90
0.30 0.10 0.60
0.80 0.60 0.40
Bias: 0.20
z values:
∑(
0.00 0.00 0.00
0.12 0.32 0.21
0.15 0.36 0.18
+
0.00 0.00 0.00
0.24 0.42 0.36
0.28 0.35 0.12
+
0.00 0.00 0.00
0.24 0.04 0.54
0.24 0.06 0.24
) + 0.2 =
2.64 4.67


Red
Green
Blue
Images:
0.00 0.00 0.00 0.00 0.00
0.00 0.20 0.40 0.30 0.00
0.00 0.30 0.90 0.60 0.00
0.00 0.90 0.10 0.20 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.60 0.60 0.90 0.00
0.00 0.40 0.70 0.40 0.00
0.00 0.70 0.50 0.30 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.00 0.00 0.00 0.00
0.00 0.80 0.40 0.90 0.00
0.00 0.30 0.10 0.60 0.00
0.00 0.80 0.60 0.40 0.00
0.00 0.00 0.00 0.00 0.00
Weights:
0.10 0.20 0.60
0.60 0.80 0.70
0.50 0.40 0.30
0.60 0.60 0.90
0.40 0.70 0.40
0.70 0.50 0.30
0.80 0.40 0.90
0.30 0.10 0.60
0.80 0.60 0.40
Bias: 0.20
z values:
∑(
0.09 0.12 0.00
0.06 0.16 0.00
0.00 0.00 0.00
+
0.42 0.24 0.00
0.20 0.21 0.00
0.00 0.00 0.00
+
0.08 0.24 0.00
0.18 0.04 0.00
0.00 0.00 0.00
) + 0.2 =
2.64 4.67 3.58
5.19 8.75 4.90
3.80 4.66 2.24

- Assume 16 kernels are used



Because we have 16 kernels

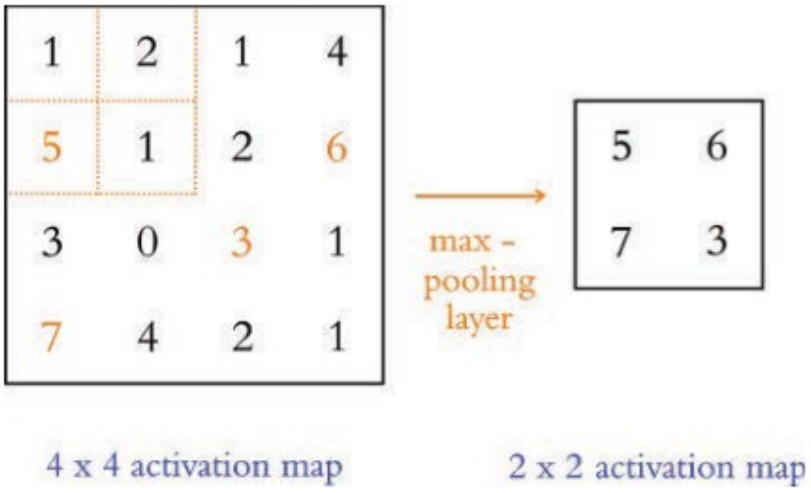
input activation map

Convolutional filter hyperparameters

- Kernel size, usually 3x3 or 5x5
- Stride length: the size of the step that the kernel moves
- Padding
- For example, consider a 28x28 image, 5x5 kernel, stride of 1
- Then the output will be 24x24
- In order to obtain a 28x28 output, we need pad the image with zeros around the edges
- In this example, we use padding of 2

Pooling layer

- This layer is to reduce the overall count of parameters
- Pooling layer usually uses the max operation, called max-pooling
- Typically, a filter size is 2x2, and stride length is 2



Example

```
model = Sequential()

# first convolutional layer:
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
                 input_shape=(28, 28, 1)))

# second conv layer, with pooling and dropout:
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())

# dense hidden layer, with dropout:
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))

# output layer:
model.add(Dense(n_classes, activation='softmax'))
```

Deepfake

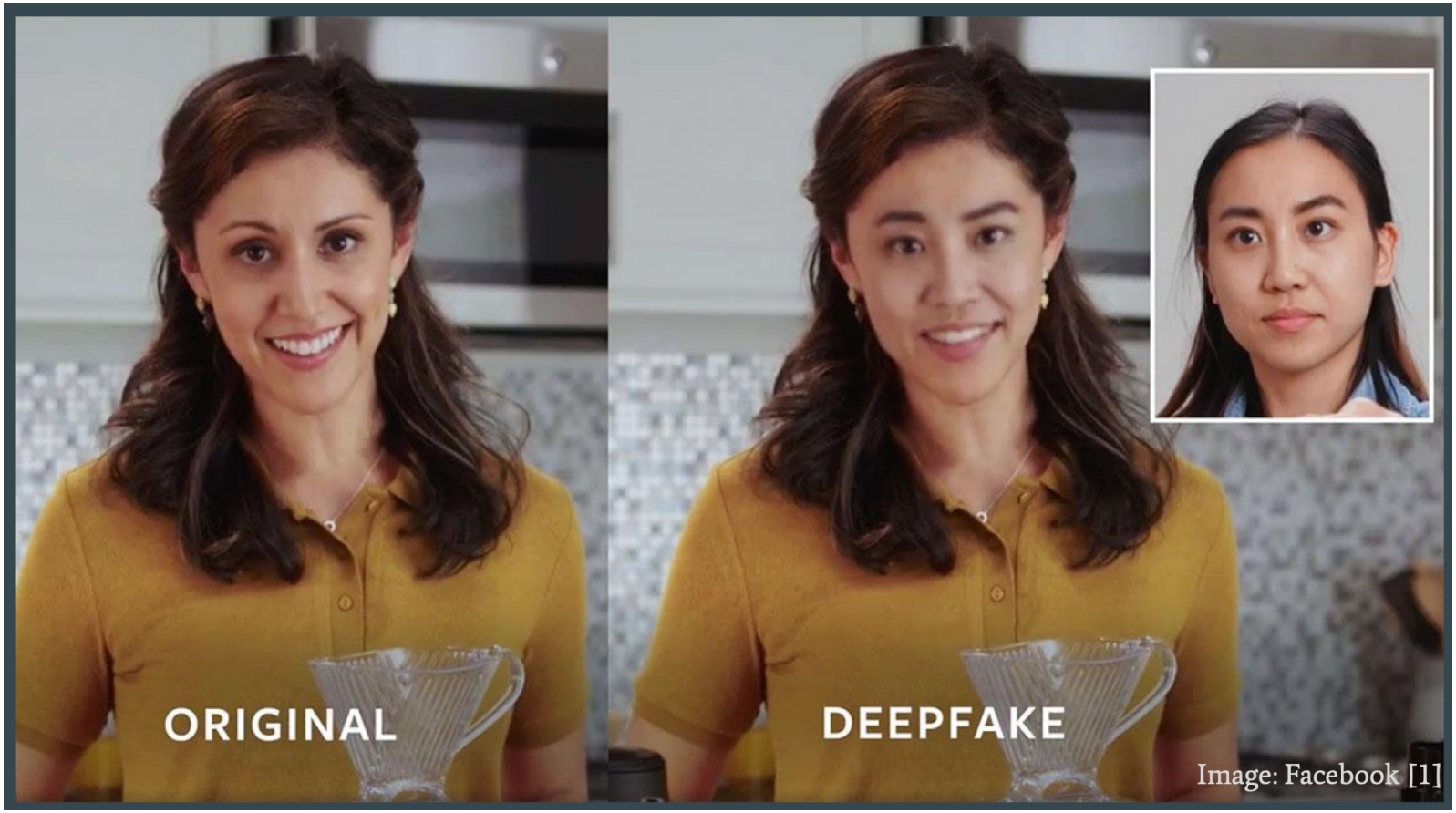
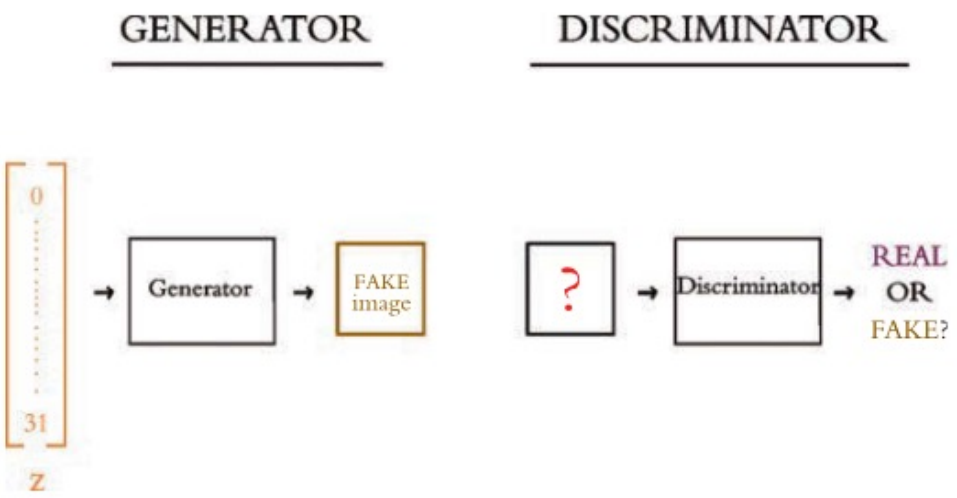


Image: Facebook [1]

Generative adversarial network

- GAN involves two deep neural networks
- One network is a generator that produces forgeries of images
- The other network is a discriminator that distinguish real or fake images

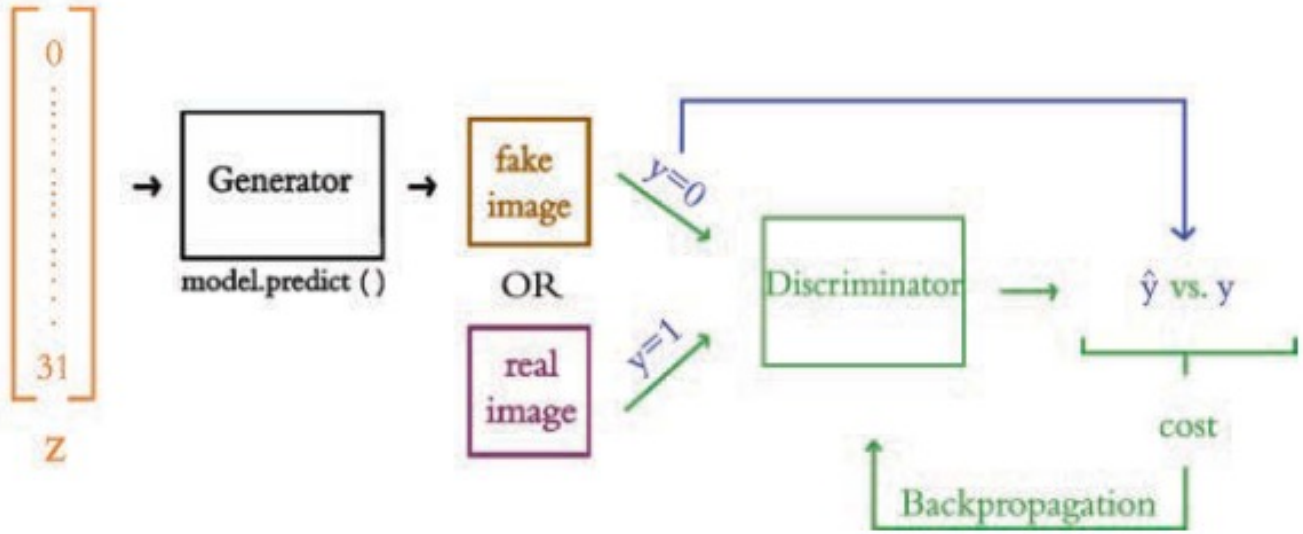


Basic theory

- Discriminator training
- In this process, the generator produces fake images, while the discriminator learns to tell the fake images from real ones
- Generator training
- The discriminator judges fake images, the generator uses this information to learn how to better fool the discriminator into classifying fake images as real ones

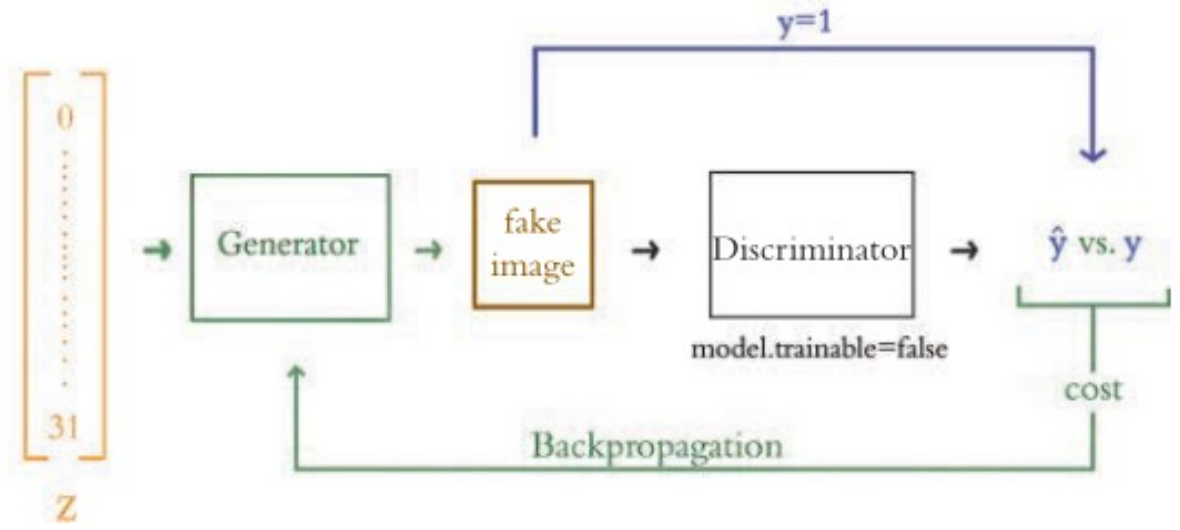
Discriminator training

- Generator produces fake images
- Discriminator outputs a prediction
- Binary classification



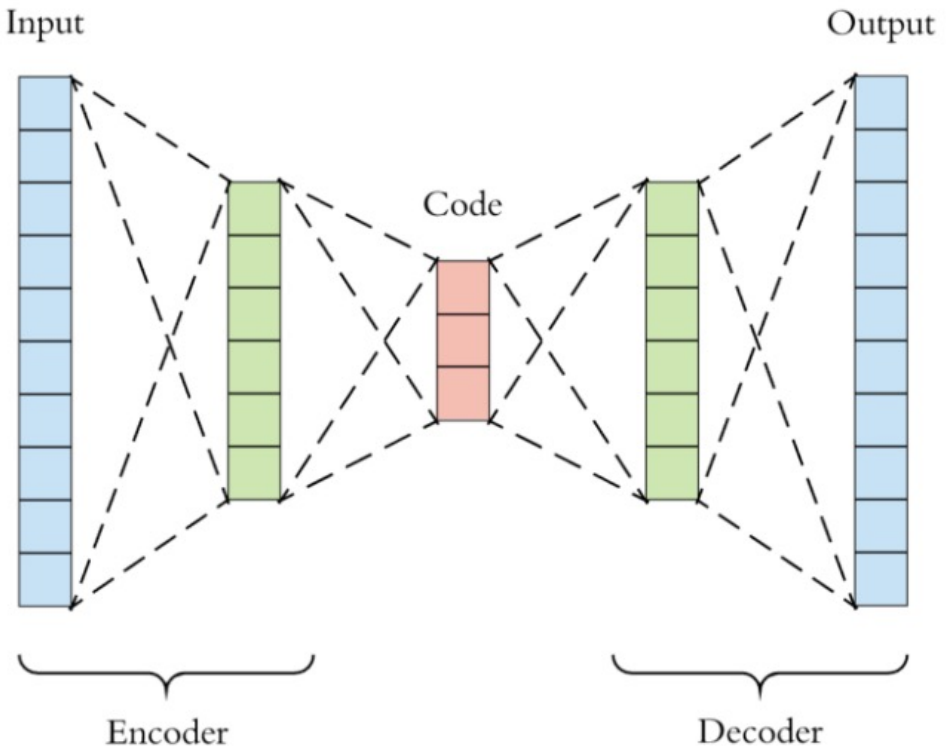
Generator training

- Generator takes a random input z
- The fake images produced are fed into the discriminator
- We lie to the discriminator, and label all of these fake images as real $(y = 1)$



Autoencoder

- It can be used to construct a reduced representation (latent space)



Latent space

Each image is represented by a point in n-dimensional (latent) space