

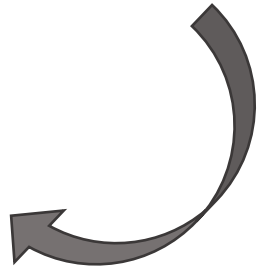
An Introduction to Neural Networks

Bjorn Burkle



<https://bit.ly/34W0fHJ>

Interactive python
notebook!



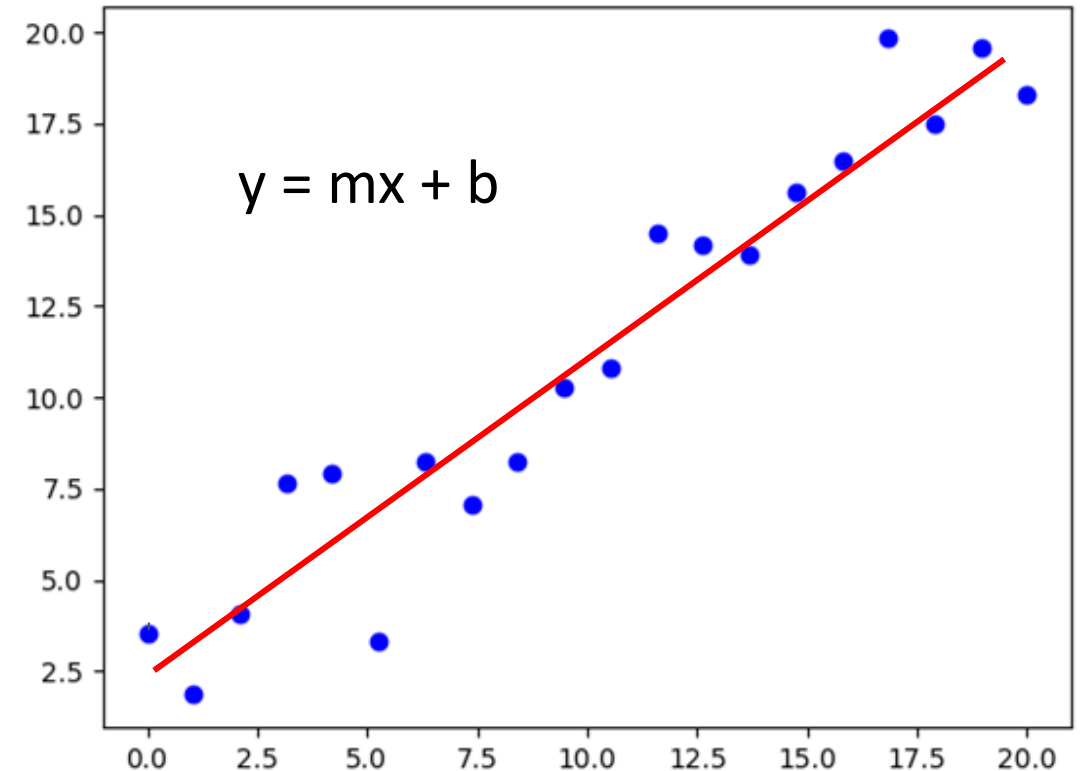
BROWN

Outline

- Traditional Fitting
- Basics of Neural Networks
 - Fully connected (dense) networks
 - Convolutional Networks
- Other common networks

Traditional Fitting

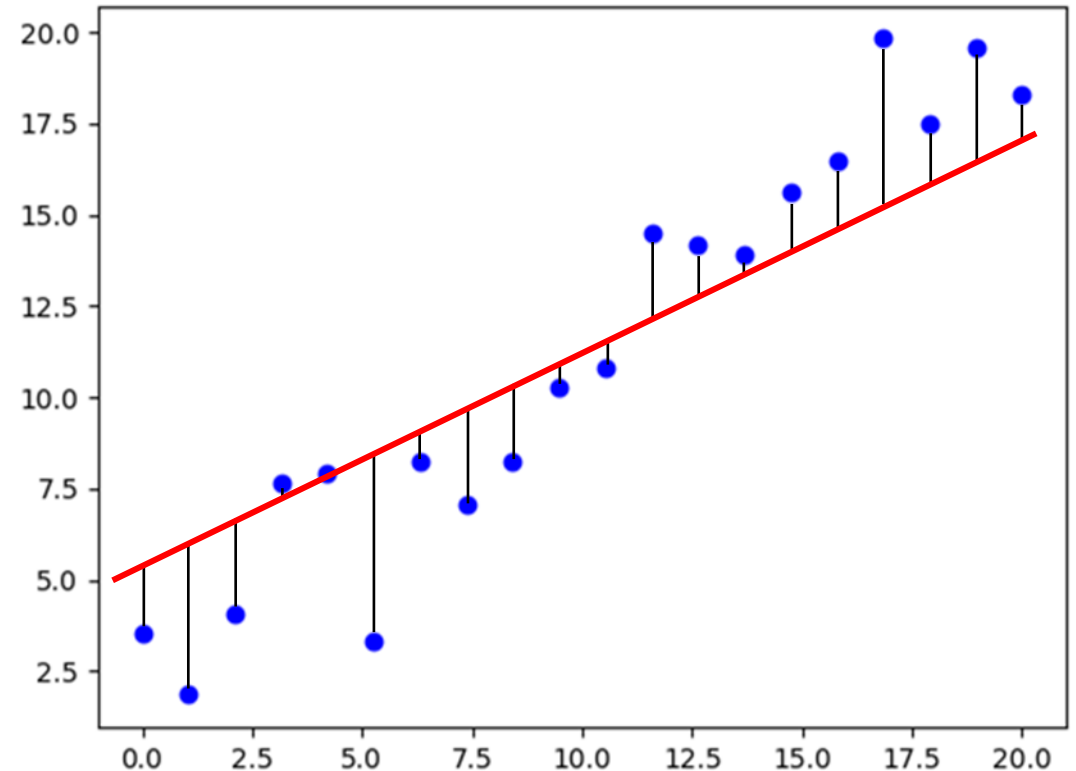
- Traditionally, we learn to fit data via linear regression
 - Start with some data
 - Give your computer an assumed function
 - Computer uses an algorithm to “learn” the best set of coefficients which fit your data
- But what is the computer the computer actually doing?





Traditional Fitting

1. Start with some data
2. Supply a function which should fit your data
 - $y = mx + b$
3. Computer makes an initial guess at coefficients
4. Computer calculates a metric which tells you how bad your fit is
 - MSE, χ^2 , etc



Traditional Fitting

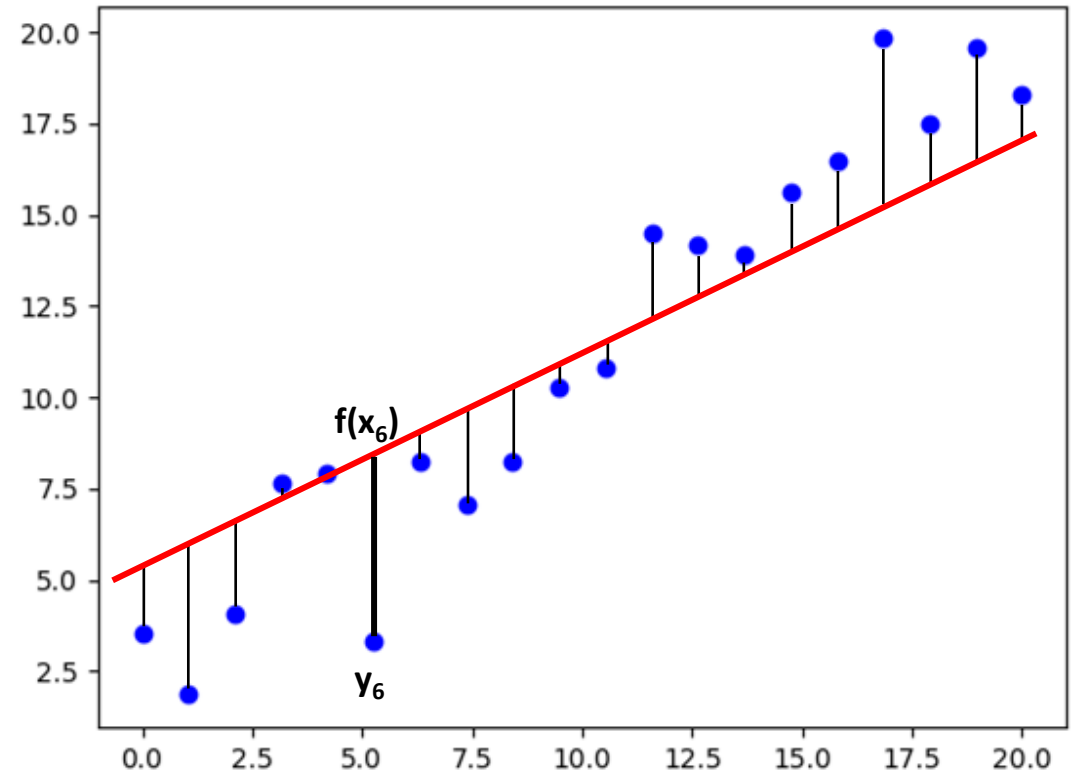
4. Computer calculates a metric which tells you how bad your fit is

- MSE, χ^2 , etc

$$MSE = \frac{1}{N} \sum_{i=1}^N (f(x_i) - y_i)^2$$

5. Computer calculates *gradient* of the metric for each variable

$$\frac{\delta(MSE)}{\delta\beta_j} = \frac{2}{N} \sum_{i=1}^N \frac{\delta f(x_i)}{\delta\beta_j} (f(x_i) - y_i)$$



Traditional Fitting

- Computer simultaneously minimizes the *gradient* for all variables

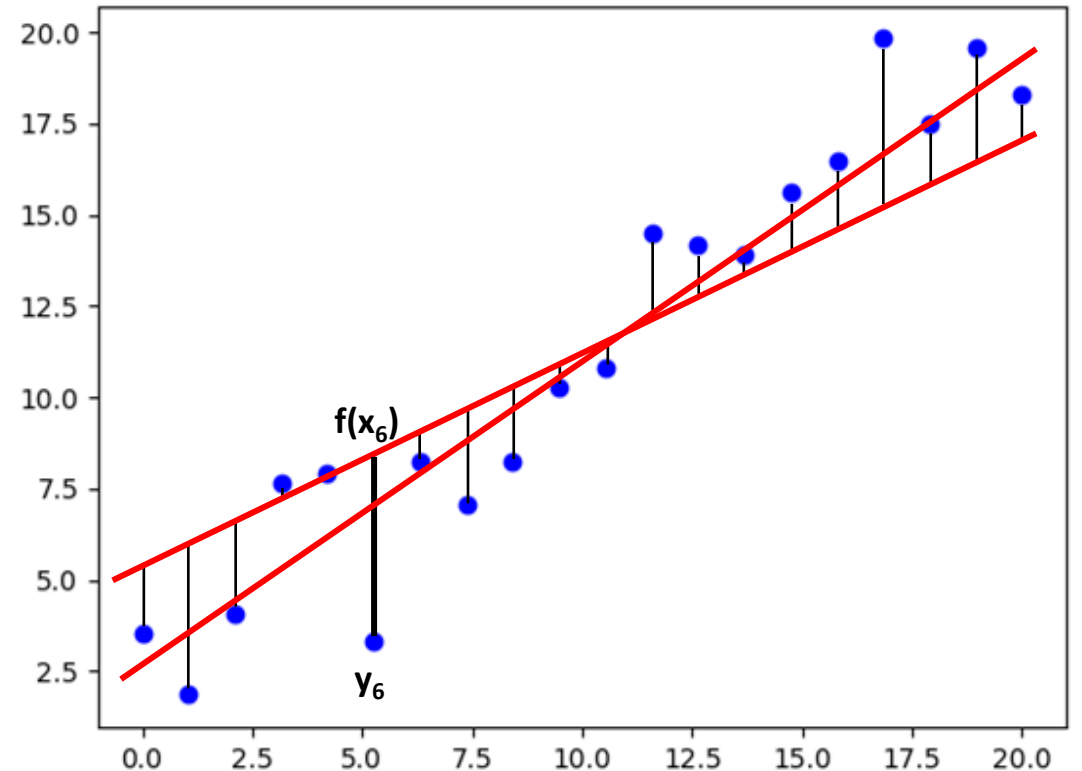
$$\frac{\delta(MSE)}{\delta m} = \frac{2}{N} \sum_{i=1}^N x_i ((mx_i + b) - y_i)$$

$$\frac{\delta(MSE)}{\delta b} = \frac{2}{N} \sum_{i=1}^N ((mx_i + b) - y_i)$$

- Adjust variables in downward direction of *gradient*

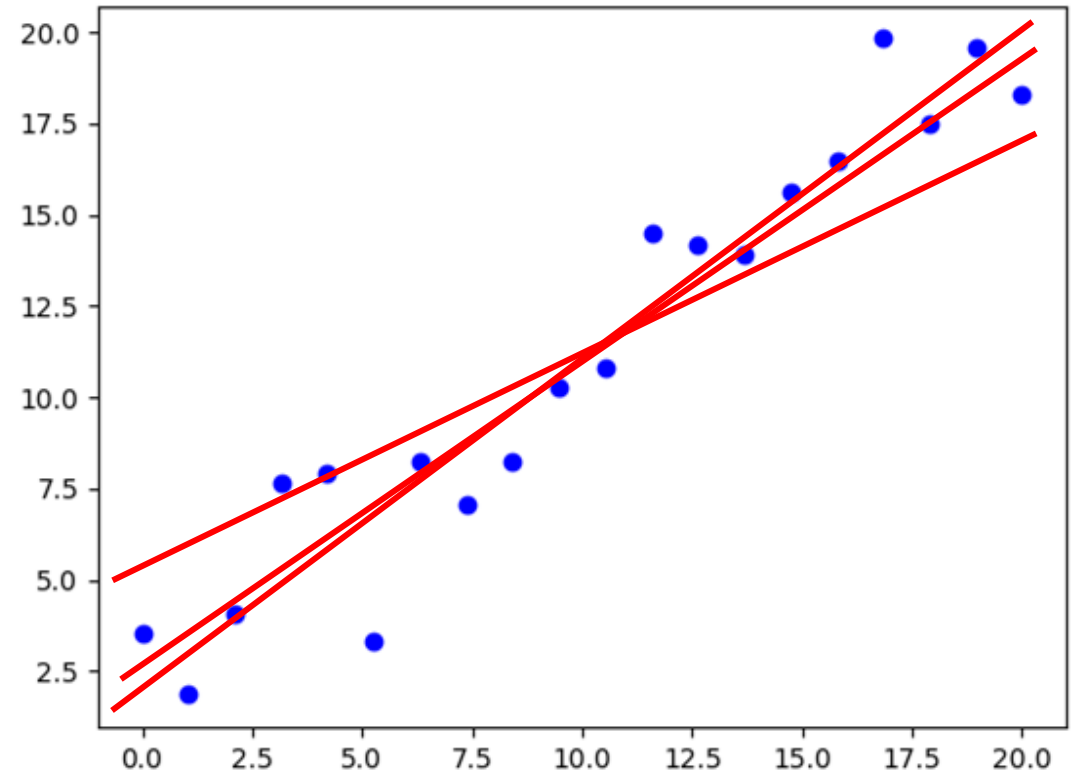
$$m^{k+1} = m^k + \Delta_m$$

$$b^{k+1} = b^k + \Delta_b$$



Traditional Fitting

8. Procedure can be iterated many times
 - Often stops once the fit falls into a minimum, i.e. the gradient stops decreasing



Traditional Fitting

Linear fitting can be broken down into the following steps:

1. Define a function which maps $F(x_i, p^u) \rightarrow y_i^{\text{pred}}$
2. Calculate a “loss” which measures how poorly y_i^{pred} predicts y_i^{true}
3. Adjust the parameters p^u to minimize your loss
4. Repeat until your loss stops decreasing

Traditional Fitting

Linear fitting can be broken down into the following steps:

1. Define a function which maps $F(x, \theta) \rightarrow y_{\text{pred}}$
2. Can this function predict y_i^{true} ?
3. Are there any functions that can predict y_i^{true} ?
4. Repeat until your loss stops decreasing.

But what if you can't trivially define a function for your problem?

Is there any hope for us then?

Linear Operators

- A linear operator is able to map a vector of inputs x_i to an output vector y_j
- We can represent all sorts of functions and transformations as a linear operator
 - Can we do the same thing when trying to understand high dimensional and abstract data sets

$$[x_0 \quad \cdots \quad x_n] * \begin{bmatrix} w_{00} & \cdots & w_{0m} \\ \vdots & \ddots & \vdots \\ w_{n0} & \cdots & w_{nm} \end{bmatrix} + b = [y_0 \quad \cdots \quad y_m]$$

Use the features of our input as a basis

Represent our unknown function as a transformation matrix

Use our possible outputs as a basis

Linear Operators

- A linear operator is able to map a vector of inputs x_i to an output vector y_j

• We can use the same set of fitting rules to construct our matrix?

- Can we do the same thing when trying to understand high dimensional and abstract data sets

$$[x_0 \quad \dots \quad x_n] * \begin{bmatrix} w_{00} & \dots & w_{0m} \\ \vdots & \ddots & \vdots \\ w_{n0} & \dots & w_{nm} \end{bmatrix} + b = [y_0 \quad \dots \quad y_m]$$

Use the features of our input as a basis

Represent our unknown function as a transformation matrix

Use our possible outputs as a basis



Well yes, but actually no

Perceptron

What we defined is called a perceptron, unfortunately it faces the drawbacks of some basic linear algebra

- The size of the matrix is defined by the length of the input and output vector
 - If x and y are scalars, you only have one coefficient to define their relationship
- A single multiplication step is not enough to encapsulate complicated relationship between variables
 - It is also not enough to encapsulate complicated forms of a function

Matrix Multiplication

- What if we do iterative multiplication?
 - Represent our function as consecutive linear operators
 - You can represent high order polynomials as a product of first order polynomials, and we know that we can represent complicated functions as polynomials via Taylor expansions

$$(((\vec{x} \times A) \times B) \times C) = \vec{y}$$

- Can increase number of variables by modifying the dimensions of B
- Multiple “internal” matrix operations can encapsulate more of the internal features of your functions

Matrix Multiplication

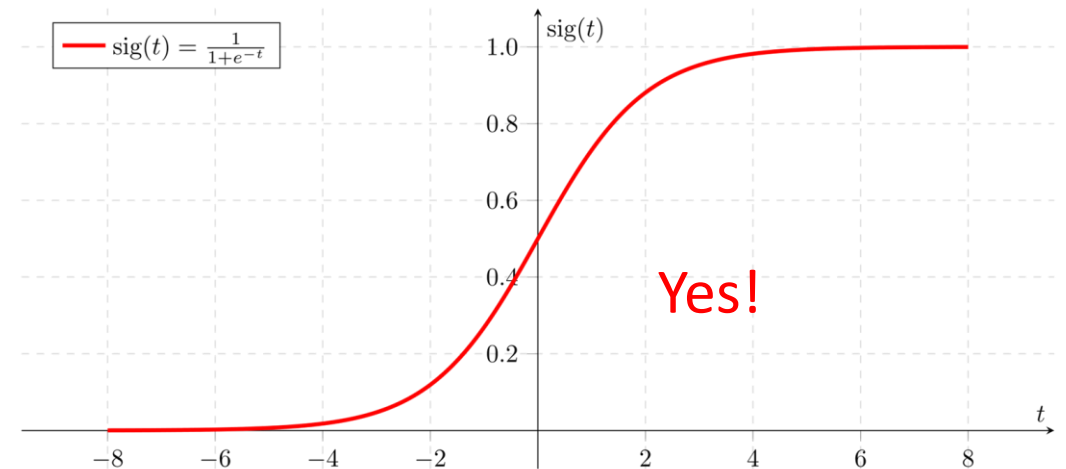
- What if we do iterative multiplication?
 - Represent our function as consecutive linear operators
 - You can represent high order polynomials as a product of first order polynomials, and we know that we can represent complicated functions as polynomials via Taylor expansions

$$\left(\left((\vec{x} \times A) \times B \right) \times C \right) = \vec{x} \times (ABC) = \vec{x} \times D = \vec{y}$$

- Can increase number of variables by modifying the dimensions of B
- Multiple “**Associativity ruins this possibility**” of the internal features of your functions

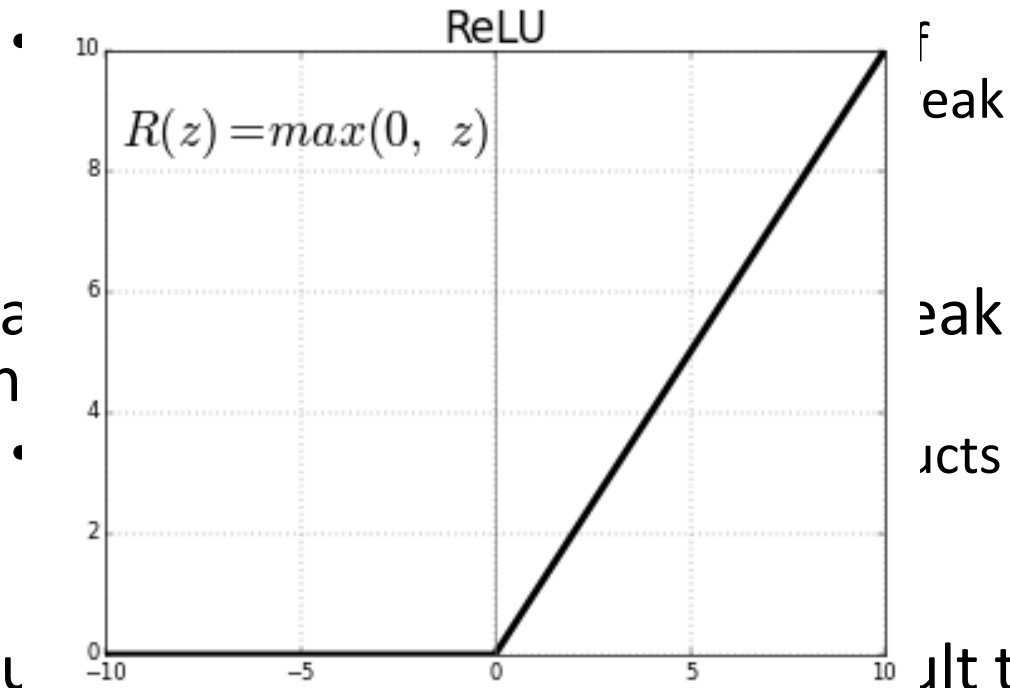
The Need for “Activation Functions”

- Is there a way to break associativity of our matrices?
 - What if we pass the output vector of each matrix through a function to break the associativity
- Can use activation functions to break this property!
 - Allows us to learn functions as products of iterative matrix multiplication
- But, it becomes increasingly difficult to calculate a non-zero gradient as you add together functions like $\text{sig}(t)$



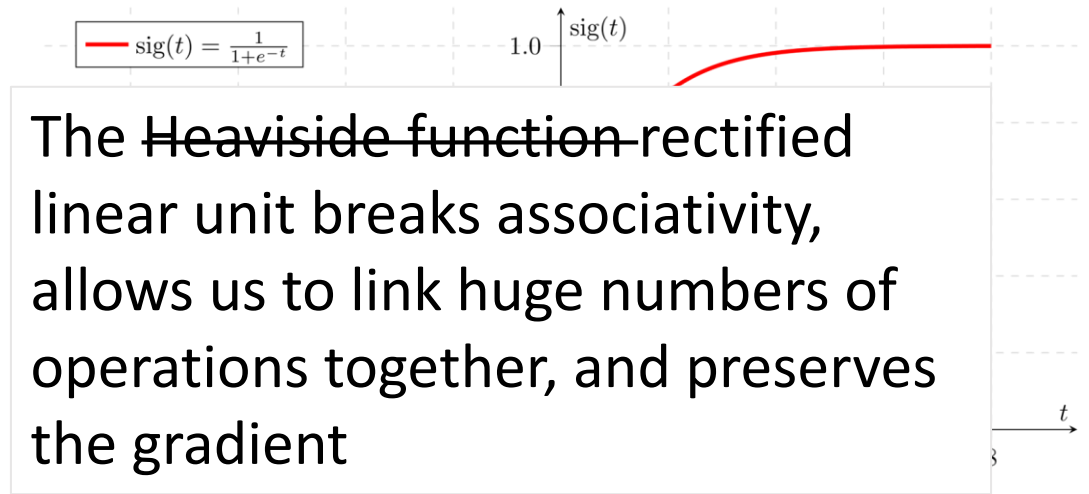
The Need for “Activation Functions”

- Is there a way to break associativity of our matrices?

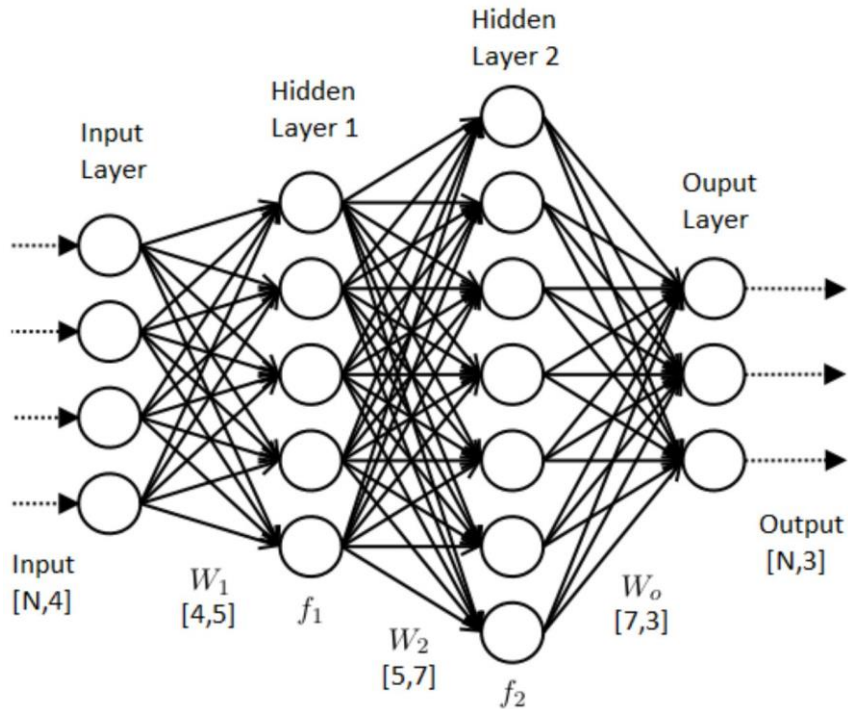


- Can

- But calculate a non-zero gradient as you add together functions like sig(t)

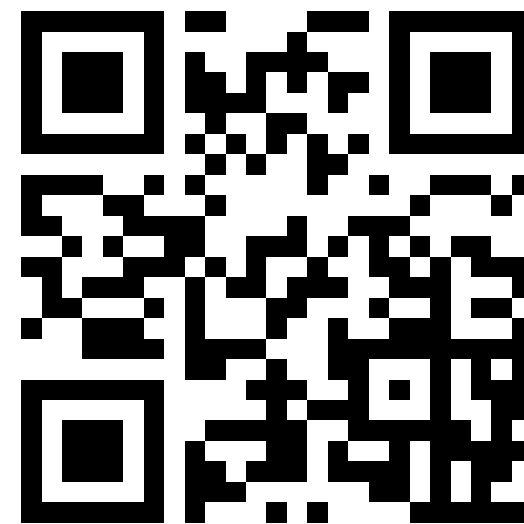


A Feed Forward Neural Network



- We just described a *feed forward* (dense) neural network
- A series of matrix multiplication operations linked together by *activation functions*
- Machine learning uses fitting algorithms to adjust the values of the *weights* inside the matrices for them to model some underlying function

Example – MNIST



<https://bit.ly/34W0fHJ>

Example – MNIST

- Task: can a computer learn to identify handwritten numbers?

- Dataset:
 - Input: 28x28 images
 - Each pixel between 0-255
 - Output: either 0, 1, 2, ..., 8, 9
 - 60k images

- Our “function” the network is learning:
 - The relationship between image pixel intensity at a given location and the number that is drawn



Example – MNIST

The overview:

1. Load the dataset
 - Want to split into a piece we *train* on and a piece we *test* performance on
 - Possibly perform some *preprocessing* to the data showing it to our network
2. Construct our *model*
3. Set some training parameters
4. Train the network
 - Feed some data through as *forward propagation*
 - Use results to modify the network via *backwards propagation*
 - Repeat for multiple *epochs* over the entire dataset
5. Can check performs on the *testing* dataset



Example – MNIST

Import your python libraries to run the code

```
[ ] import os, sys
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds # needed to download mnist
import time
```

Create your datasets

```
[ ] (ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)
```

Task: Can a computer learn how to identify handwritten numbers?



Example – MNIST

This block is just retrieving the data for us, and setting up some basic parameters

```
[ ] (ds_train, ds_test), ds_info = tfds.load(  
    'mnist',  
    split=['train', 'test'],  
    shuffle_files=True,  
    as_supervised=True,  
    with_info=True,  
)
```

Telling tensorflow the name of the dataset we want to load

Telling tensorflow that we want to load the pre-split training and testing set

Randomize the order in which the images appear

In addition to the image, also give us the label telling us what the number is

Returns ds_info which is just some metadata about the dataset

Example – MNIST

```
def normalize_img(image, label):  
    ...  
    data type of image starts as an 8-bit int, but networks work best when  
    inputs are normalized to max values ~1 (depends on type of data)  
    So, we want to make a function which transforms our image pixels to  
    go from range [0,255] -> [0., 1.]  
    ...  
    return tf.cast(image, tf.float32) / 255., label  
  
# Construct I/O pipeline for training set  
ds_train = ds_train.map(  
    # map -> before sending batches to GPU your CPU  
    # will perform transformation function  
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)  
ds_train = ds_train.cache()  
ds_train = ds_train.shuffle(ds_info.splits['train'].num_examples)  
ds_train = ds_train.batch(128)  
ds_train = ds_train.prefetch(tf.data.AUTOTUNE)  
  
# Construct same I/O pipeline for testing set  
ds_test = ds_test.map(  
    normalize_img, num_parallel_calls=tf.data.AUTOTUNE)  
# Note: Don't need to shuffle testing set  
ds_test = ds_test.batch(128)  
ds_test = ds_test.cache()  
ds_test = ds_test.prefetch(tf.data.AUTOTUNE)  
  
--NORMAL--
```

map will apply a function to your dataset as it's being loaded into memory

- `num_parallel_calls` is telling your computer how many images it should do this for at a time

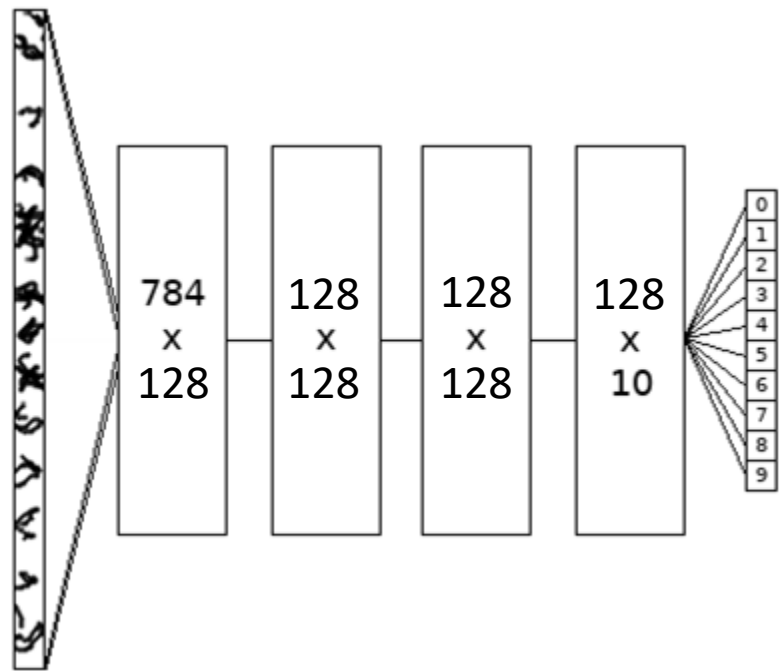
shuffle is saying that you want the computer to re-arrange the order in which the images are fed into the network per *epoch*

batch is saying that when changing your weights, you first want to calculate the changes for 128 images and average those changes

prefetch tells the network how many images you want to pre-load when feeding them into the network

Example - MNIST

```
# A sequential model is constructed
# These are very nice if you have a "1D" model where each
# layer feeds directly into the next
dense_model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```



Constructing the network architecture

- Unravels the 28x28 images into a 1D array
- 2 *hidden* layers inside of the network
- Uses a ReLU activation function between each layer
- Outputs to a vector of length 10
 - Each element is the networks confidence of the number being that specific digit

Example - MNIST

Setting some parameters for training

```
dense_model.compile(  
    optimizer=tf.keras.optimizers.Adam(0.001),  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()],  
)
```


- **Optimizer:** An algorithm which tells our network how much it should adjust its weights when training
- **Loss:** The function which our network is trying to minimize
- **Metrics:** Other functions which we want the network to show us so we can monitor its performance

Categorical Cross Entropy

$$\text{Loss} = - \sum_{i=0}^9 y_i^{\text{true}} * \log y_i^{\text{pred}}$$

- y_i^{pred} = network's output value for that digit
- $y_i^{\text{true}} = 1$ if the real value, otherwise 0

Example – MNIST

```
dense_model.fit(  
    ds_train,  
    epochs=10,  
    validation_data=ds_test,  
)
```

This function does all the training and validation for us

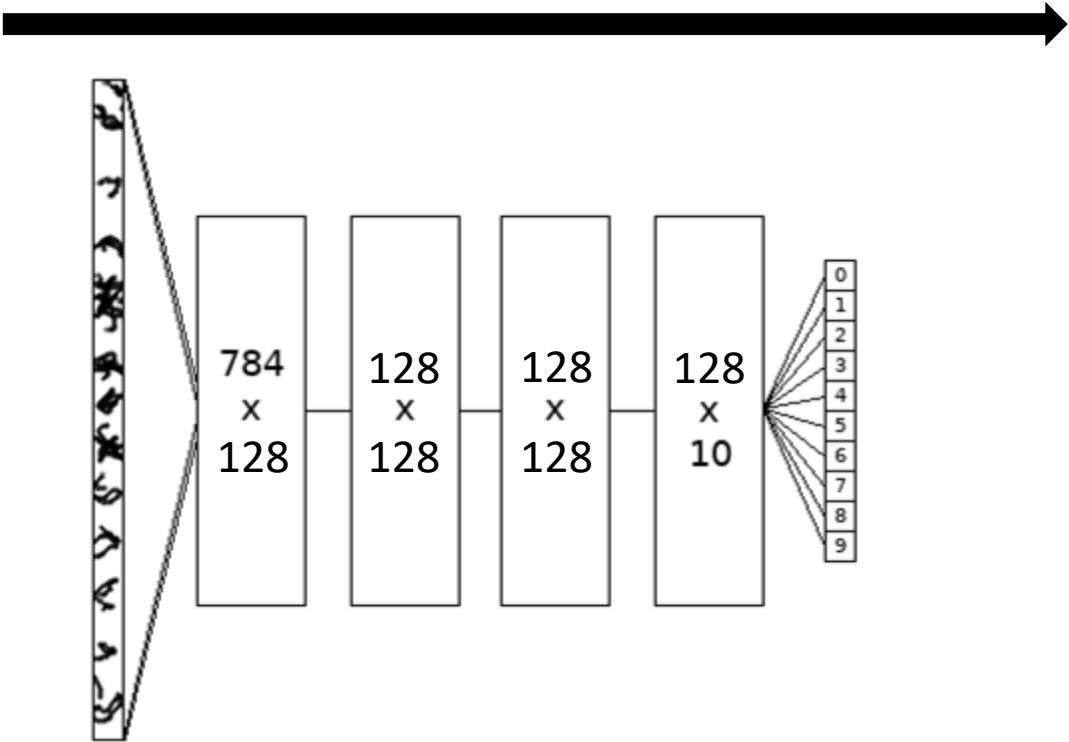
- *Forward propagation*
- *Backward propagation*
- Runs for 10 epochs

Example – MNIST

```
▶ dense_model.fit(  
  ds_train,  
  epochs=10,  
  validation_data=ds_test,  
)
```

This function does all the training and validation for us

- *Forward propagation*
- *Backward propagation*
- Runs for 10 epochs



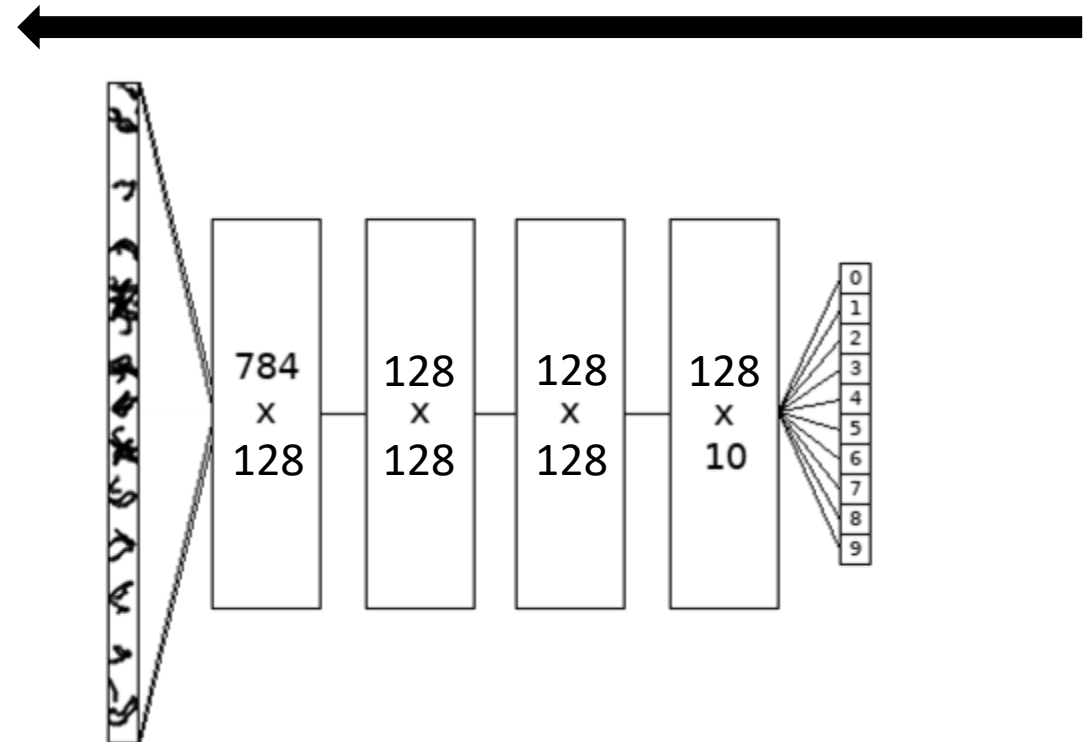
1. Network feeds the images through the network
2. Calculates the loss for each image

Example – MNIST

```
▶ dense_model.fit(  
  ds_train,  
  epochs=10,  
  validation_data=ds_test,  
)
```

This function does all the training and validation for us

- *Forward propagation*
- ***Backward propagation***
- Runs for 10 epochs

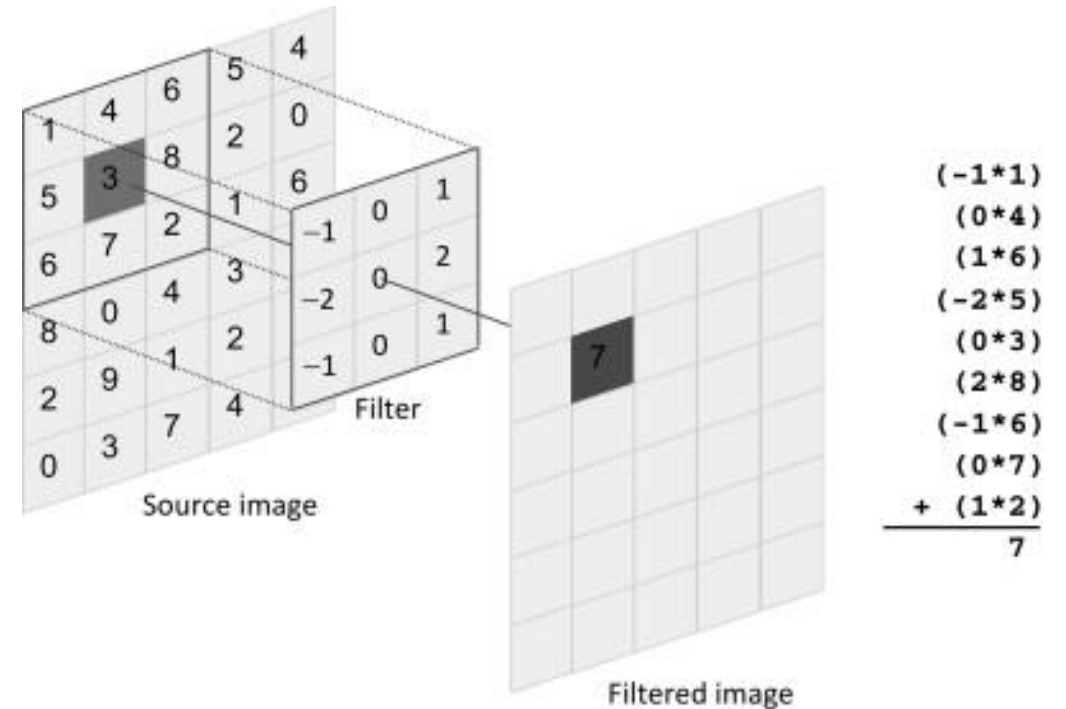


1. Network calculates the gradient associated with the loss of each image (via chain rule)
2. Adjusts weights in each matrix (based on loss and optimizer)

Convolutional Networks

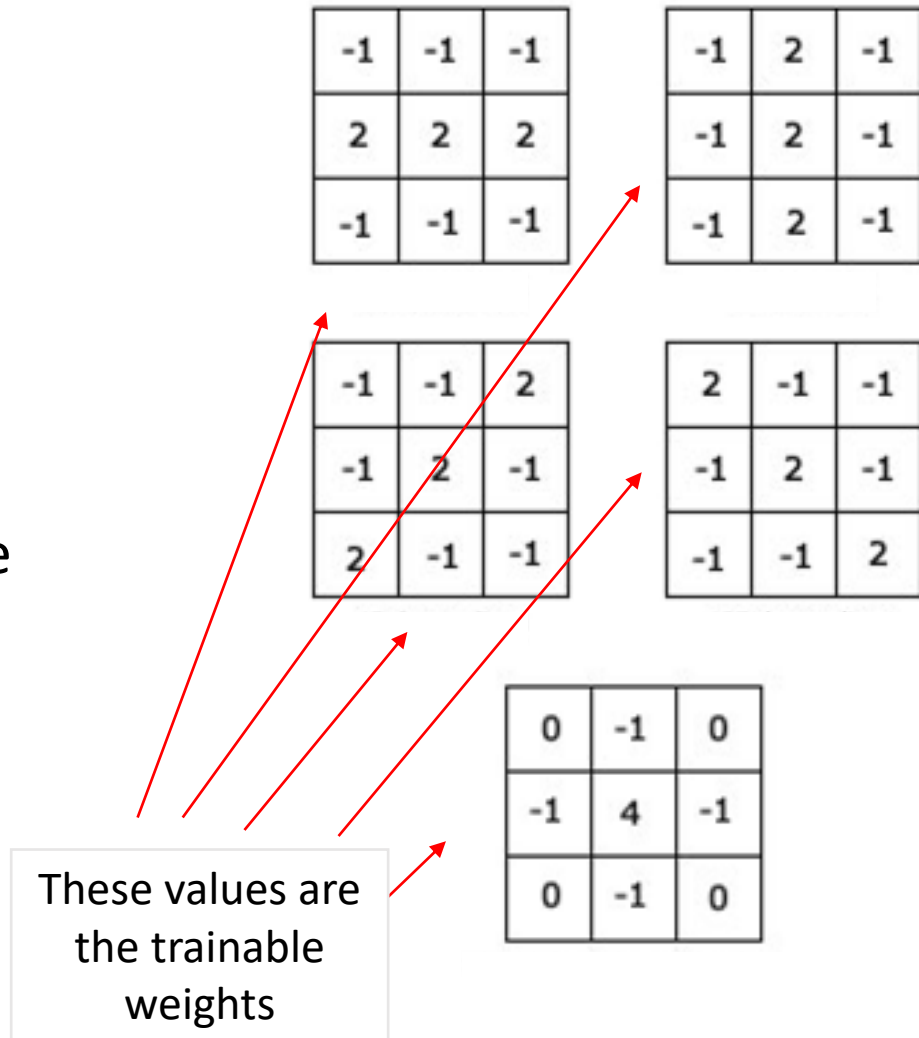
Convolution Matrices

- Use small matrices to extract nearest-neighbor features from your inputs
 - Matrices are then convoluted across the input
- Each network *layer* consists of multiple *filters*
 - Can be 1D, 2D, or 3D matrices
 - **Kernel Size** is the dimensions of the matrix
 - **Stride** is the distance the matrices move along inputs between extraction steps



Convolution Matrices

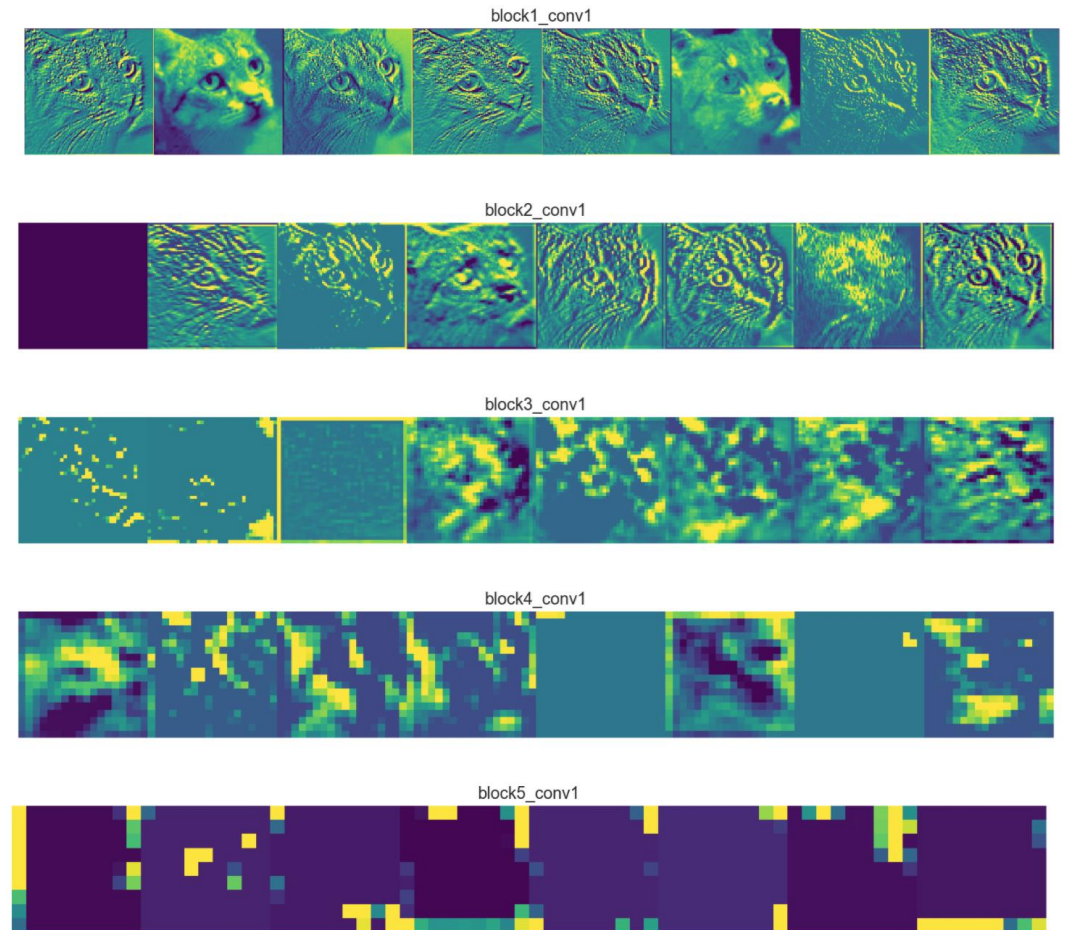
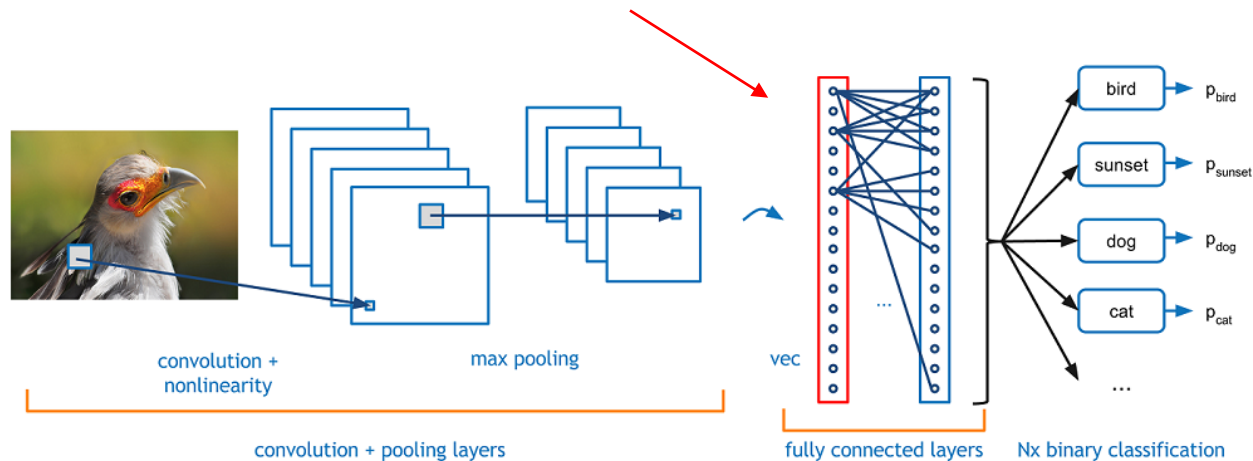
- Very good at extracting important features from an image
 - But also any dataset where order of variables is important!
- Nothing new, used in standard image recognition for a long time
 - Machine learning is used to have the algorithm decide what features to extract
 - Allows us to iteratively apply convolutions to extract increasingly abstract features



Convolutional Neural Networks

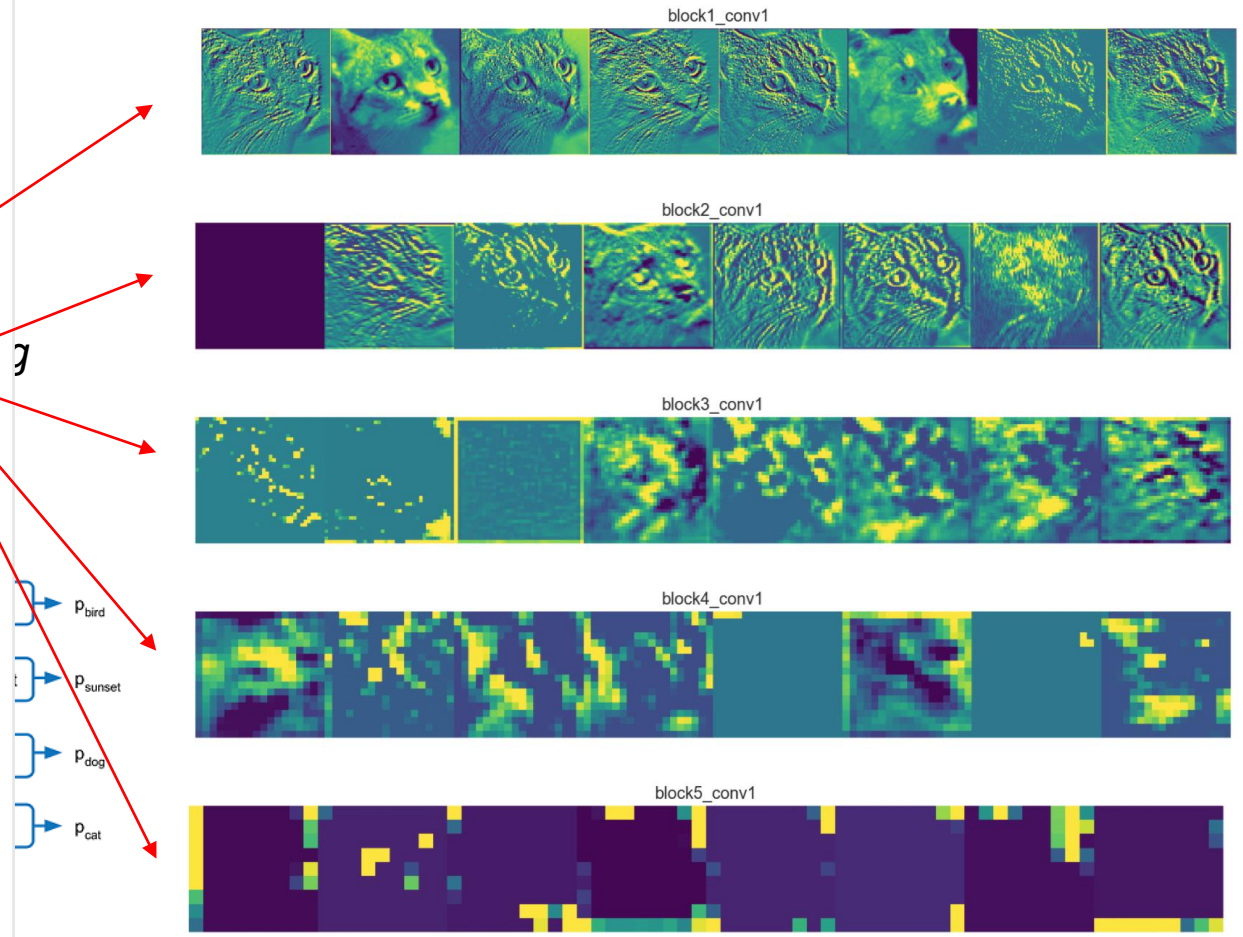
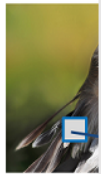
Standard CNN formation

- Use multiple layers of convolutions
 - A single layer will consist of many filters
 - Layer outputs will be multiple feature-extracted compressed “images”
 - Compression based on *stride* and *pooling*
- Final convolution layer fed into a *fully connected* layer (big matrix)



Convolutional Neural Networks

- **S** Way to think about what's happening
 - Input data, which is an array of input features, is not always in the best representation of your data
 - Each filter transforms the input image into a new feature basis
 - The more convolution layers, the more abstract basis you are transforming your original inputs into
 - In QM, we are used to writing our wave function in the orthonormal basis of our Hamiltonian
 - This trivializes the problem, and is not a strategy unique to QM
 - If our network wants to model an unknown function, can represent it as a single matrix if our data is transformed into its corresponding orthonormal basis



γ

P_{bird}

P_{sunset}

P_{dog}

P_{cat}

classification

CNN Example - MNIST

```
[ ] def custom_CNN(input_shape=(28,28,1)):
    ...
    Creates a network infrastructure using a functional form
    This is good if you want more control over the network architecture
    Additionally, some architectures might connect layers in more
    create ways
    ...

    inputs = tf.keras.layers.Input(input_shape)
    print(inputs.shape)

    x = tf.keras.layers.Conv2D(16, (3, 3), strides=(2,2), padding='same')(inputs)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.MaxPool2D(pool_size=(2,2))(x)
    print(x.shape)

    x = tf.keras.layers.Conv2D(32, (2, 2), strides=(1,1), padding='same')(x)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.MaxPool2D(pool_size=(2,2))(x)
    print(x.shape)

    x = tf.keras.layers.Conv2D(64, (2, 2), strides=(1,1), padding='same')(x)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.GlobalMaxPool2D()(x)
    print(x.shape)

    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(10)(x)
    x = tf.keras.layers.Softmax()(x)

    return tf.keras.Model(inputs=inputs, outputs=x)
```

Using the functional API for making networks in TensorFlow

1. Define an input layer
2. Pass input through a *2D convolution*
 - (N layers, (x and y filter dimension), (x and y movement), 'pad edges of image with 0 to match dimension')
 - Pass output feature-extracted images through relu
 - Use a local pool to further compress images (decrease RAM usage)
3. 2 more sets of Conv2D
 - Each time compress images and use more filters
4. Use a *Global Pool* to compress each of the final 64 abstract feature images into a single value
 - Flatten these into a 1D array of length 64
5. Pass through dense network
6. Pass output values through *softmax*
7. Convert into a TensorFlow Model object

CNN Example - MNIST

```
[ ] def custom_CNN(input_shape=(28,28,1)):
    ...
    Creates a network infrastructure using a functional form
    This is good if you want to create a network in a functional form
    Additionally, some architectures can be created using the functional API
    ...

    inputs = tf.keras.layers.Input(input_shape)
    print(inputs.shape)

    x = tf.keras.layers.Conv2D(64, (2, 2), strides=(1,1), padding='same')(x)
    x = tf.keras.layers.ReLU()(x)
    x = tf.keras.layers.GlobalMaxPool2D()(x)
    print(x.shape)

    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(10)(x)
    x = tf.keras.layers.Softmax()(x)

    return tf.keras.Model(inputs=inputs, outputs=x)
```

Using the functional API for making networks in TensorFlow

1. Define an input layer

2. *2D convolution*

(kernel dimension), (x and y dimension) of image with 0 to match

3. Extract images through relu and max pooling to compress images (decrease resolution)

4. Flatten images and use more filters

5. Compress each of the final images into a single value

6. Flatten these into a 1D array of length 64

5. Pass through dense network

6. Pass output values through *softmax*

7. Convert into a TensorFlow Model object

Softmax

- Transform each value in output vector to

$$p_i = \frac{e^{-l_i}}{\sum e^{-l_j}}$$

- l is the raw output of the i -th element in the array
- Basically calculating a probability given the raw output values by using a partition function

CNN Example - MNIST

```
[ ] cnn_model = custom_CNN()

# Unlike before, can't automatically associate them to
# the network using compile
cnn_loss = tf.keras.losses.SparseCategoricalCrossentropy()
cnn_optimizer = tf.keras.optimizers.Adam(0.001)
train_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
val_acc_metric = tf.keras.metrics.SparseCategoricalAccuracy()
```

- Can now call the function to construct your network
- For this example we will be “manually” running a training loop
 - Must define our loss function, optimizer, and metrics as individual variables
 - Will not be associating with network in TensorFlow backend using compile

CNN Example - MNIST

Performing training loop manually rather than using built-in functions

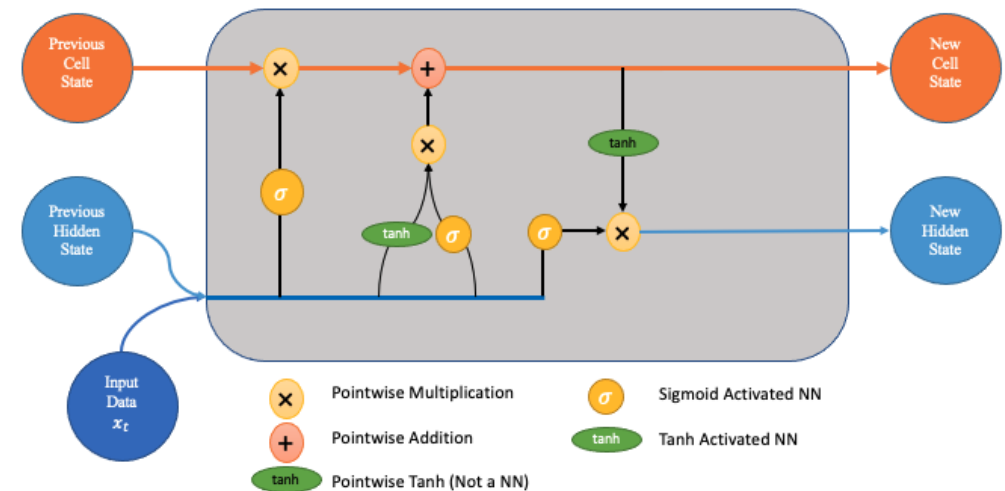
- Running over training set
 - Forwards propagation
 - Backwards propagation
 - Checking metrics on testing set
- Evaluating over testing set
 - Only forward pass
 - Keep track of accuracy and loss for each batch
 - Check accuracy and loss after finishing
- Repeat for desired number of epochs

```
[ ] def TrainLoop(epochs):  
  
    for i in range(epochs):  
        print('Starting epoch:', i+1)  
        start = time.time()  
  
        for step, (x_batch, y_batch) in enumerate(ds_train):  
            with tf.GradientTape() as tape:  
                y_pred = cnn_model(x_batch, training=True)  
                pred_loss = cnn_loss(y_batch, y_pred)  
  
            grads = tape.gradient(pred_loss, cnn_model.trainable_variables)  
            cnn_optimizer.apply_gradients(zip(grads, cnn_model.trainable_variables))  
            train_acc_metric.update_state(y_batch, y_pred)  
  
            if step % 50 == 0:  
                print(' >> Training on batch', step)  
  
        train_acc = train_acc_metric.result()  
        print('Finished training epoch', i+1)  
        print(' >> Training accuracy: %.4f' % train_acc)  
        print(' >> Took %.2f seconds' % (time.time()-start))  
        train_acc_metric.reset_states()  
  
        # After each epoch, manually run over network  
        # with validation dataset  
        start_val = time.time()  
        pred_loss_val = []  
        print('Validating network...')  
        for step, (x_batch, y_batch) in enumerate(ds_test):  
  
            # when validating make sure the network is run with training off  
            y_pred_val = cnn_model(x_batch, training=False)  
            pred_loss_val.append(cnn_loss(y_batch, y_pred_val).numpy())  
            train_acc_metric.update_state(y_batch, y_pred_val)  
  
            # No need to perform gradient decent when validating  
  
        if step % 50 == 0:  
            print(' >> Running on batch', step)  
  
        train_acc = train_acc_metric.result()  
        print('Finished running over testing dataset')  
        print(' >> Validation accuracy: %.4f' % train_acc)  
        print(' >> Validation loss: %.4f' % (sum(pred_loss_val)/len(pred_loss_val)))  
        print(' >> Took %.2f seconds' % (time.time() - start_val))  
        train_acc_metric.reset_states()
```

Other Common Networks

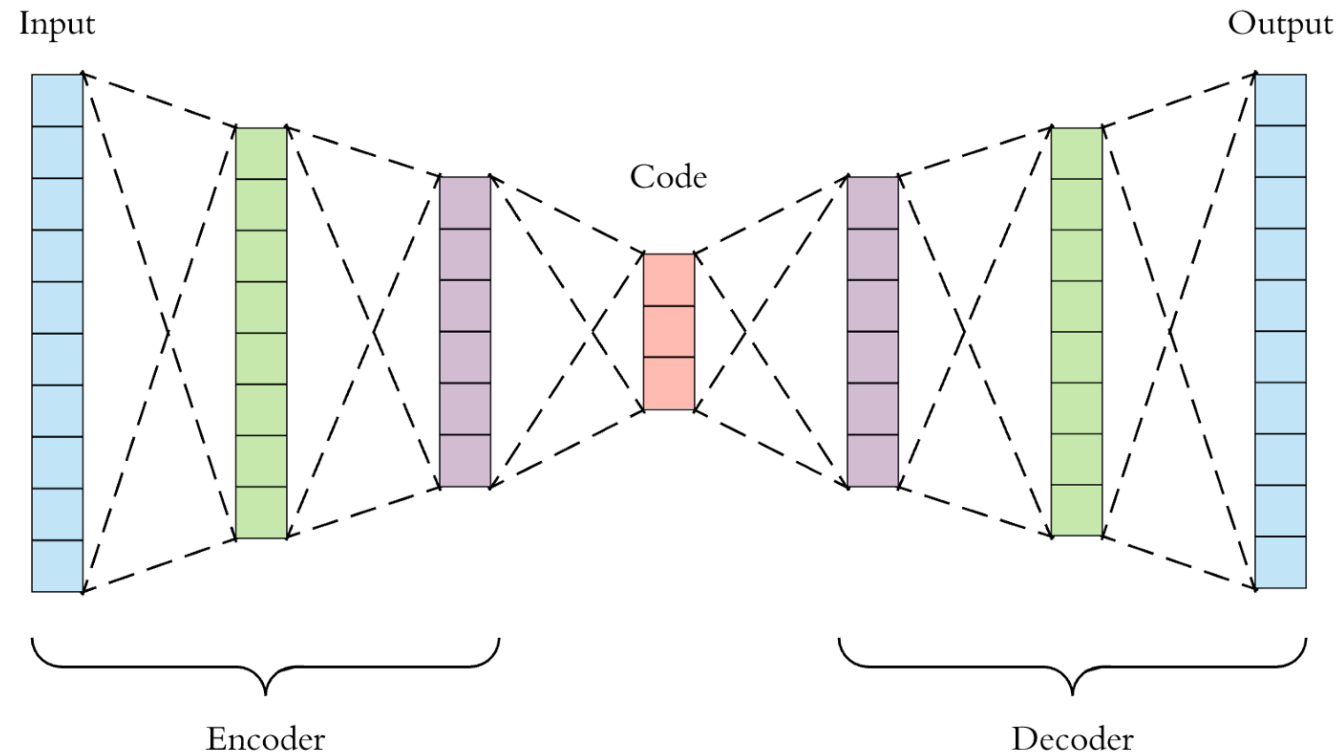
RNNs (Long Short-term Memory)

- In ordered data, it can be important to know what came before previous data points
 - e.g. your phone predicting the next word you are going to type
- Convolutions can work, but will have problems with variable length input arrays
 - e.g. number of words in a sentence
 - Can also use LSTM to regularize data of variable length before feeding into later network layers
 - e.g. data is an array of molecules, each with 6 features (position and momentum)
 - If numbers of molecules isn't the same in each datapoint, can pass through LSTM layer before a fully connected layer



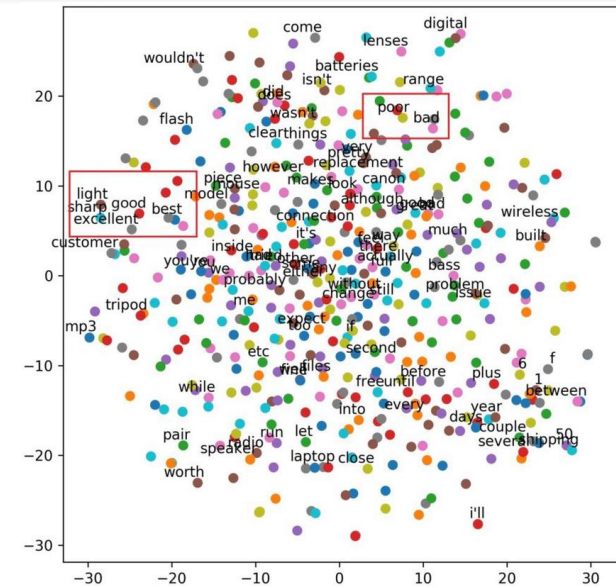
Auto Encoders

1. Use first half of network to compress your input into a smaller dimensionality
2. Use second half of network to decompress back to the original size
3. If the output matches the input, then the network learned to encode the input information in a low dimensional *latent space*



Auto Encoders – Uses

- **Unsupervised learning**
 - Auto-encoders don't need labeled data, but will still map similar inputs to similar spaces in latent space
 - Can use clusters in latent space to classify data post-training
- **Word embeddings** - <https://bit.ly/3sGrLku>
 - Use auto-encoder on words
 - Input is a very long array of length $O(\text{dictionary})$, 1 if it is the word, 0 if not
 - Latent space is array of 50 floats
 - Allows computer to map words to a high dimensional vector space
 - Used as the first step in all levels of language processing
- **Defect detection**
 - Pass images through an auto-encoder trained on normal photos
 - If there is something wrong (e.g. slight photoshopping marks) that wasn't in training data, network won't know how to encode or decode it properly
 - When looking at output, the *defects* will be made more pronounced

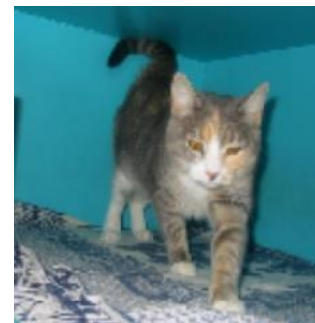


Generative Networks

Train a network to produce something new

- Use two networks
 - One to generate datapoints
 - One to differentiate real from generated data points
- Generative text
 - See [GPT-2](#) or GPT-3
 - Takes insane amount of training data
- Image generation
- Data simulation
 - MC generation can be computationally intensive and slow, running a pre-trained network is generally quite fast

```
File Edit Selection View Go Run Terminal Help
en.json def sortByKey(key, array) Untitled-1
1 def sortByKey(key, array) |:
  for i in range(0, len(array)) :
    for j in range(i, len(array)) :
      if array[i][key] < array[j][key] :
        array[i], array[j] = array[j], array[i]
  return array
2
```



Real



Fake

Closing Remarks – A Cautionary Message

Telling the network to “learn the function” can be dangerous

- It can be hard for us to gain intuition about what it is learning
- The network might not be learning what we want it to
 - Ex: Neural networks don’t identify stop signs by being red octagons
 - If there is a bias in your data, the networks will happily exploit that to “cheat”
- Machine learning is **very** inefficient
 - Ex: Have to look at $O(10000)$ images of numbers multiple times to know how to differentiate them
 - My opinion: Nothing intelligent about “Artificial Intelligence”

Despite this, neural networks are very powerful tools

- With great power comes great responsibility
 - Ex: social media and targeted ads
 - The strongest computer algorithms in the world are actively being used to exploit human dopamine responses to get them addicted to social media and sell them products