Python/Tensorflow2.0/Keras Cheat Sheet

Cheat sheet made for useful info and links about the above mentioned subjects. Follows the order of the Pierian Data course that I am taking/took.

```
ANN - Artificial Neural Network

CNN - Convolutional Neural Network

RNN- Recurrent Neural Network

GAN- Generative Adversarial Networks
```

NUMPY CRASH COURSE (Section 3):

Notes:

https://colab.research.google.com/drive/1FQf03eB61a6XV8uvA6vJr5ht70l-mt2y#scrollTo=aqPPigawgoLM

Worksheet:

https://colab.research.google.com/drive/1mFVFMvLmx7ykgajZMa0nme8iB-DHAHai

PANDAS CRASH COURSE (Section 4):

Notes:

https://colab.research.google.com/drive/1fps7mg5_exR_TFb6bfYi2svmo8d33q0i#scrollTo=ZvrdQ75bTLOY

PANDAS OPERATIONS:

https://colab.research.google.com/drive/1nT1LlJ9iMXBkWO1ATqnjgJ0UI1sv2FOD

https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

Missing Data Notes:

https://colab.research.google.com/drive/1QxNCwFfq2plqIvlTn2Iedi0VL_oEPecz#scrollTo=9GmAtaz62ogX

GroupBy:

https://colab.research.google.com/drive/1ExIKSSHP2rqovHsArAvX7AN3z1DckcqN#scrollTo=MbnFVnYu4mqa

IO:

https://colab.research.google.com/drive/1-JvKQIeKDJeZSkd2ayjAsWYUTrbl6qga#scrollTo=Zw1kb2xjTIiS

WorkSheet:

https://colab.research.google.com/drive/1bf86u0XXn9R8dG6LDE4csirvqF2_6eWW

Data Visualization Crash Course (Section 5):

Matplotlib Basics:

https://colab.research.google.com/drive/1hoQFqoLVaRNnhGqFKPyXPV6GwU6VKoad#scrollTo=kwvJiKuKecxB

Seaborn Basics:

https://colab.research.google.com/drive/1nmgZO_E2knj0NlhkUHq4nM8xbLUpTh-Y

Worksheet:

Machine Learning Concepts Overview (Section 6):

Machine Learning:

Machine learning is a method of data analysis that automates analytical model building. Iteratively learn from data.

Two types of machine learning tasks:

Supervised Learning: Algs trained using labeled examples. Such as input where desired output is known. Example label: Spam vs. Legitimate email examples.

The network receives inputs with the corresponding correct outputs. The alg learns by comparing its outputs with the correct ones to find errors. It then modifies based on the comparison.

Commonly used historical data likely predicts future events.
Acquire->Clean->Model&Train->Test->Deploy

3 Sets of Training Data:

-*Training Data*: Used to train the model parameters

-*Validation Data*: Used to determine what model Hyperparameters to adjust

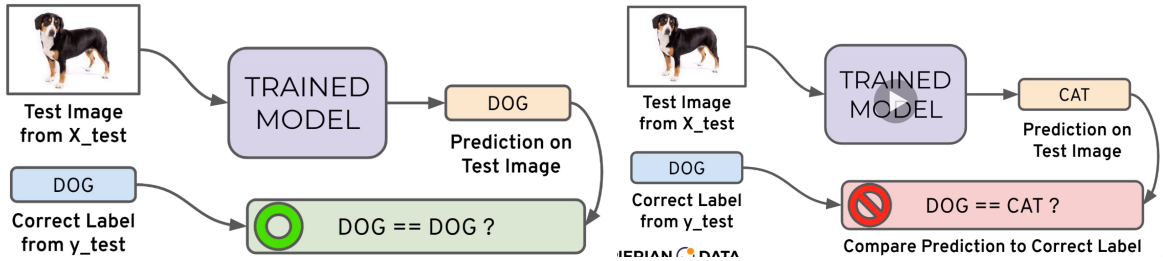
-(FINAL MODEL)*Test Data*: Used to get some final performance metric

Overfitting and Underfitting:

Overfitting: The model fits too much to the noise from the data. Results in low error on training set but high error on test/validation sets.

Underfitting: The model does not capture the underlying trend of the data and does not fit the data well enough. Results in low variance but high bias. Underfitting is often a result of an excessively simple model.

Evaluating Performance - Classification Error Metrics:



Accuracy: number of correct predictions made by the model divided by the total number of predictions. A good metric in balanced data sets (50/50 split of dogs and cats for example). Ex: 8/10 correct.

TP = True Positive / TN = True Negative / FN = False Negative

Recall: ability to find all of the relevant cases within a dataset. I.E: TP/(TP+TN) = Recall

Precision: Ability of a classification model to identify only the relevant data points. TP/(TP + FN)

F1-Score: Optimal blend of precision and recall we can combine the two metrics using what is called the F1-Score. The harmonic mean of precision and recall taking both metrics into account in the following equation. **F1 = 2*((precision * recall)/(precision + recall)).**

Evaluating Performance - Regression Error Metrics:

Regression is a task when a model attempts to predict continuous values. Example: Attempting to predict the price of a house given its features is a **regression task**. VS>>> Predicting the country a house is in given its features would be a **classification task.**

-Mean Absolute Error (MAE):

Abs val of errors. $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$ Won't punish large errors.

-Mean Squared Error (MSE):

Mean of the squared errors. Larger errors are noted more than with MAE making MSE more popular. $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

-Root Mean Square Error (RMSE):

Most popular. Has the same units as Y and punishes large values

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

Confusion Matrix:

Confusion Matrix

		predicted condition	
	total population	prediction positive	prediction negative
true condition	condition positive	**True Positive (TP)**	**False Negative (FN)** (type II error)
	condition negative	**False Positive (FP)** (Type I error)	**True Negative (TN)**

Confusion Matrix

		predicted condition		
	total population	prediction positive	prediction negative	Prevalence $= \frac{\Sigma \text{ condition positive}}{\Sigma \text{ total population}}$
true condition	condition positive	**True Positive (TP)**	**False Negative (FN)** (type II error)	True Positive Rate (TPR), Sensitivity, Recall, Probability of Detection $= \frac{\Sigma \text{ TP}}{\Sigma \text{ condition positive}}$
	condition negative	**False Positive (FP)** (Type I error)	**True Negative (TN)**	False Positive Rate (FPR), Fall-out, Probability of False Alarm $= \frac{\Sigma \text{ FP}}{\Sigma \text{ condition negative}}$
	Accuracy $= \frac{\Sigma \text{ TP} + \Sigma \text{ TN}}{\Sigma \text{ total population}}$	Positive Predictive Value (PPV), Precision $= \frac{\Sigma \text{ TP}}{\Sigma \text{ prediction positive}}$	False Omission Rate (FOR) $= \frac{\Sigma \text{ FN}}{\Sigma \text{ prediction negative}}$	Positive Likelihood Ratio (LR+) $= \frac{\text{TPR}}{\text{FPR}}$
		False Discovery Rate (FDR) $= \frac{\Sigma \text{ FP}}{\Sigma \text{ prediction positive}}$	Negative Predictive Value (NPV) $= \frac{\Sigma \text{ TN}}{\Sigma \text{ prediction negative}}$	Negative Likelihood Ratio (LR−) $= \frac{\text{FNR}}{\text{TNR}}$

Unsupervised Learning:

Lack of labelling...IE we don't have the correct answer for historical data. Which means evaluation is much harder and more nuanced.

Clustering:

Grouping together unlabeled data points into categories / clusters. Data points are assigned to a cluster based on similarity.

Anomaly Detection:

Attempts to detect outliers in a dataset. Ex: Fraudulent transactions on a credit card.

Dimensionality Reduction:

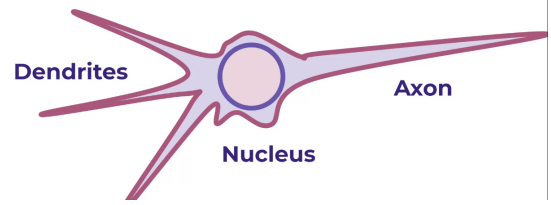
Reduces the number of features in a data set, either for the compression, or to better understand underlying trends within a data set.

Neural Networks:

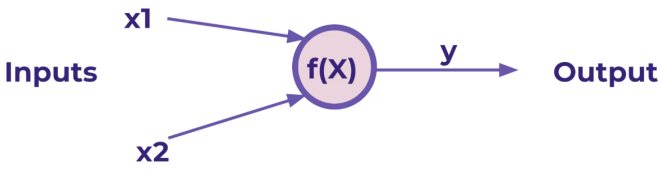
Ways of modelling biological neuron systems mathematically. These systems can be used to solve tasks that many other algs cannot. DEEP LEARNING = neural network w/ more than 1 hidden layer.

Perceptron Model:

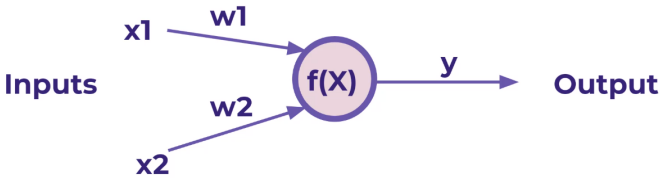
- Simplified Biological Neuron Model



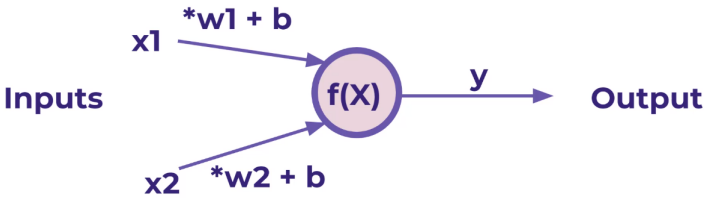
- If f(X) is just a sum, then y=x1+x2



- Now **y = x1w1 + x2w2**



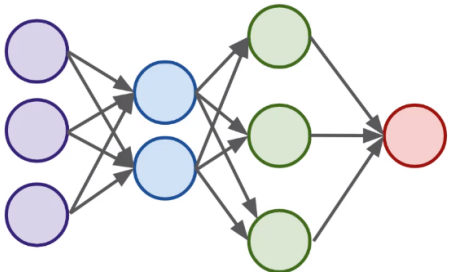
- Let's add in a **bias** term **b** to the inputs.



- y = (x1w1 + b) + (x2w2 + b)

$$\hat{y} = \sum_{i=1}^{n} x_i w_i + b_i$$

- To build a network of perceptrons, we can connect layers of perceptrons, using a **multi-layer perceptron model**.



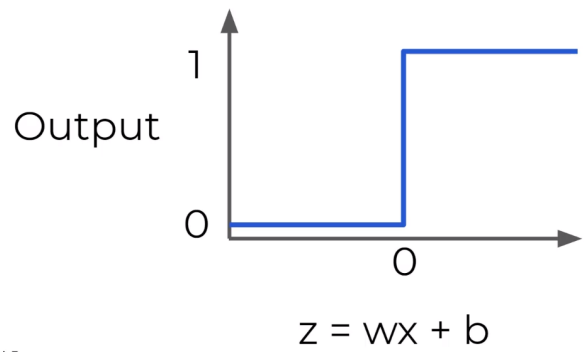
ERIAN DATA

The output of one perceptron is the input of another. This allows the network as a whole to learn about interactions and relationships between features. (Input Layer -> Hidden Layers(almost black box like)->Output Layer). A NN becomes a deep NN when there are 2 or more hidden layers.

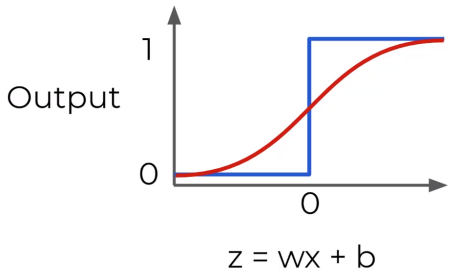
Activation Function:

Inputs x have weight w and a bias term b attached to them. IE: x*w + b. W is the importance of the input. B is used as a minimum threshold needed to overcome for the value.

We can state z = x*w + b then pass x through some activation function to limit its value.



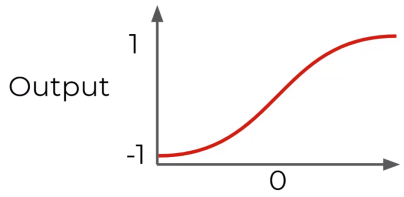
Sigmoid function (red line):



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

Hyperbolic Tangent: Outputs -1 and 1 instead of 0 and 1

- Hyperbolic Tangent: tanh(z)

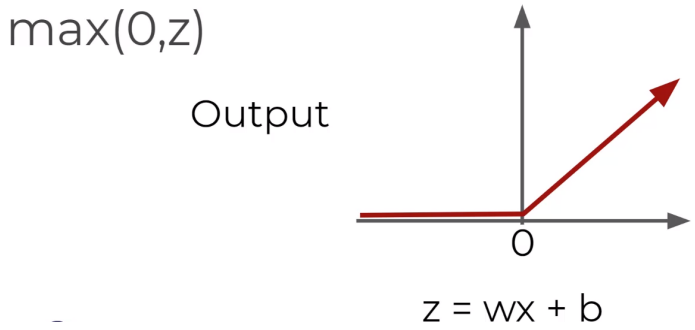


$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

Rectified Linear Unity (ReLU): This is a relatively simple function: max(0,z)



Multi-Class Classification Considerations:

2-types of multi class situations.

-Non Exclusive (A data point can have multiple classes / categories assigned to it).

- Non-Exclusive Classes

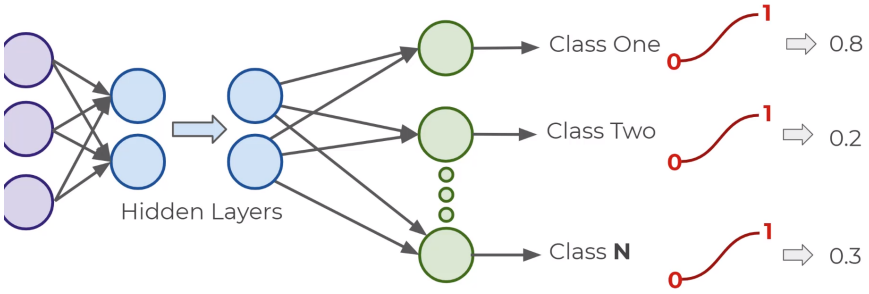
Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

→

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

Use the Sigmoid Function for non exclusive classes since each neuron will output a value between 0 and 1. Indicating the prob of having that class assigned to it.

- Sigmoid Function for Non-Exclusive Classes



-Mutually Exclusive (Only one class per data point). Ex: Photos being in grayscale or color

- Organizing Multiple Classes
  - Instead we use **one-hot encoding**
  - Let's take a look at what this looks like for mutually exclusive classes.

Data Point 1	RED
Data Point 2	GREEN
Data Point 3	BLUE
...	...
Data Point N	RED

	RED	GREEN	BLUE
Data Point 1	1	0	0
Data Point 2	0	1	0
Data Point 3	0	0	1
...	...	...	...
Data Point N	1	0	0

What is done when each data point can only have a single class assigned to it? We use the softmax function.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K$$

K = # events. This function calculates the prob of each target class over all possible target classes. Range from 0 -> 1. The sum of all the probs will be equal to 1.

- [Red , Green , Blue]
- [0.1 , 0.6 , 0.3]

Cost Functions and Gradient Descent:

NN takes in inputs,multiplies them by weights, and adds biases to them. This result is passed through an activation function which at the end of all layers leads to some output.

The output y^ is the model's estimation of what it predicts the label to be. Afterwards, how do we evaluate it? After that how can we update the network's weights and biases.

Cost function/loss function is kept track during training to monitor network performance.

- We'll use the following variables:
  - y to represent the true value
  - a to represent neuron's prediction
- In terms of weights and bias:
  - w*x + b = z
  - Pass z into activation function σ(z) = a

- One very common cost function is the quadratic cost function:

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2$$

- **W** is our neural network's weights, **B** is our neural network's biases, $\mathbf{S}^r$ is the input of a single training sample, and $\mathbf{E}^r$ is the desired output of that training sample.

$$C(W, B, S^r, E^r)$$

Adaptive Gradient Descent (Adam) to solve minimum cost problem. Step size needs to be small enough to find the minimum. IE a correct learning rate.

- When dealing with these N-dimensional vectors (tensors), the notation changes from **derivative** to **gradient**.
- This means we calculate **∇C(w1,w2,...wn)**

- For classification problems, we often use the **cross entropy** loss function.
- The assumption is that your model predicts a probability distribution p(y=i) for each class i=1,2,...,C.

- For a binary classification this results in:

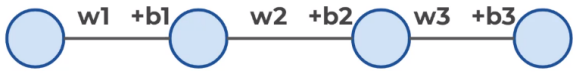
$$-(y\log(p)+(1-y)\log(1-p))$$

- For **M** number of classes > 2

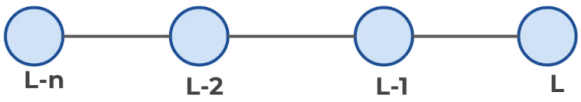
$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$$

Backpropagation:

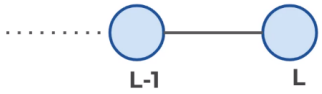
- Each input will receive a weight and bias



- Let's say we have **L** layers, then our notation becomes:



- Focusing on these last two layers, let's define **z=wx+b**
- Then applying an activation function we'll state: **a = σ(z)**



- This means we have:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$
  - $C_0(\ldots) = (a^L - y)^2$

- Using the relationships we already know along with the chain rule:

$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \frac{\partial a^L}{\partial z^L} \frac{\partial C_0}{\partial a^L}$$

- Step 2: For each layer, compute:
  - $z^L = w^L a^{L-1} + b^L$
  - $a^L = \sigma(z^L)$

- Step 3: We compute our error vector:
  - $\delta^L = \nabla_a C \odot \sigma'(z^L)$
    - $\nabla_a C = (a^L - y)$
    - **Expressing the rate of change of C with respect to the output activations**

- Step 3: We compute our error vector:
  - $\delta^L = (a^L - y) \odot \sigma'(z^L)$
- Now let's write out our error term for a layer in terms of the error of the next layer (since we're moving backwards).
- Font Note: **l**owercase **L**
- Font Note: Number **1**

- Step 4: Backpropagate the error:
  - For each layer: L−1,L−2,... we compute (note the lowercase L (l)):
    - $\delta^l = (w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$
    - $(w^{l+1})^T$ is the transpose of the weight matrix of **l+1** layer

- Step 4: When we apply the transpose weight matrix, $(w^{l+1})^T$ we can think intuitively of this as moving the error backward through the network, giving us some sort of measure of the error at the output of the lth layer.

- Step 4: We then take the Hadamard product $\odot\sigma'(z^l)$. This moves the error backward through the activation function in layer **l**, giving us the error δ^l in the weighted input to layer **l**.

- The gradient of the cost function is given by:
  - For each layer: L−1,L−2,... we compute

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \frac{\partial C}{\partial b^l_j} = \delta^l_j$$

PRETTY CONFUSING, HERE'S AN EXTERNAL LINK FOR MORE INFO:

http://neuralnetworksanddeeplearning.com/chap2.html

Keras Syntax Basics:

Choosing an optimizer and loss

Keep in mind what kind of problem you are trying to solve:

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')
```

https://colab.research.google.com/drive/1SMM09nLmUXHrGQpvCyvrVmYQcnMsBJcR

Keras Regression Code Along:

https://colab.research.google.com/drive/1kg50v5KbnBLx31TTvIRU-GE7cBTARV4z

Keras Classification Code Along - EDA and Preprocessing:

- Early Stopping
  - Keras can automatically stop training based on a loss condition on the validation data passed during the model.fit() call.
- Dropout Layers
  - Dropout can be added to layers to "turn off" neurons during training to prevent overfitting.

https://colab.research.google.com/drive/1cEMybpz6cLnESW8b-rgg0fx0MKppYjNo#scrollTo=HP_AW1nyZ24t

Keras Loan Project Solution:

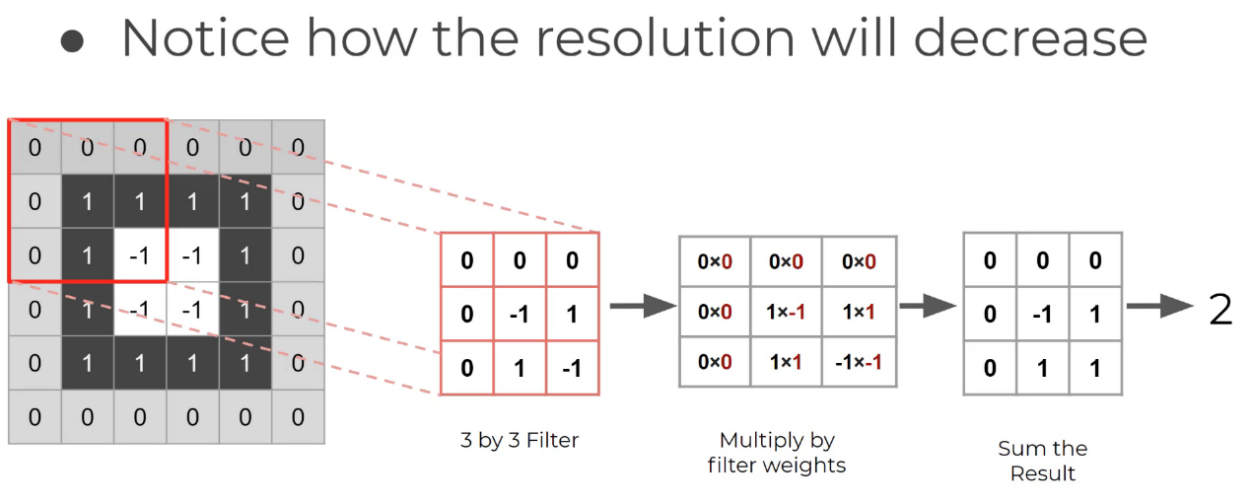
https://colab.research.google.com/drive/182vLMsd7dReuLpfpFGCgW1WR9UBHD9CG#scrollTo=Lh2njkkjZGLi

TensorBoard:

https://colab.research.google.com/drive/11tfk2F3vcFVfAkT21_NM2GIPzDBvKz0G#scrollTo=hYszfsNXq1zp

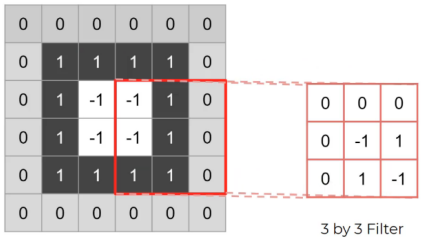
Introduction To CNN's (Convolutional Neural Networks):

Filters are essentially image kernels. Which are small matrices applied to an entire image.



- In the context of CNNs, these "filters" are referred to as **convolution kernels**.
- The process of passing them over an image is known as **convolution**.
- Let's go over a few more important factors!

We can **pad** the image with more values



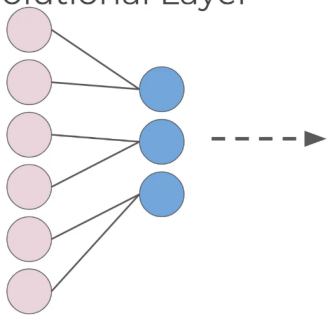
So we don't lose image size.

- ANNs
  - Large amount of parameters (over 100,000 for tiny 28 by 28 images)
  - We lose all 2D information by flattening out the image
  - Will only work on very similar, well centered images.

- A CNN can use convolutional layers to help alleviate these issues.
- A convolutional layer is created when we apply multiple image filters to the input images.
- The layer will then be trained to figure out the best filter weight values.

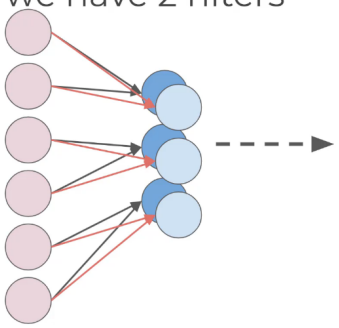
- A CNN also helps reduce parameters by focusing on **local connectivity.**
- Not all neurons will be fully connected.
- Instead, neurons are only connected to a subset of local neurons in the next layer (these end up being the filters!)

- CNN - Convolutional Layer



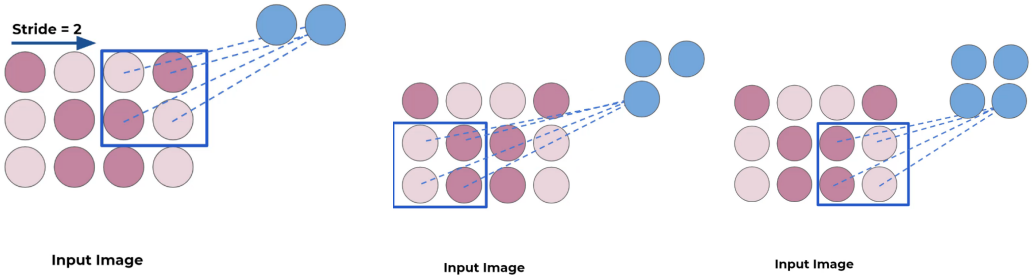
ERIAN DATA

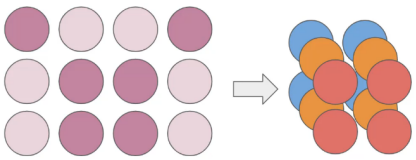
- CNN - Here we have 2 filters



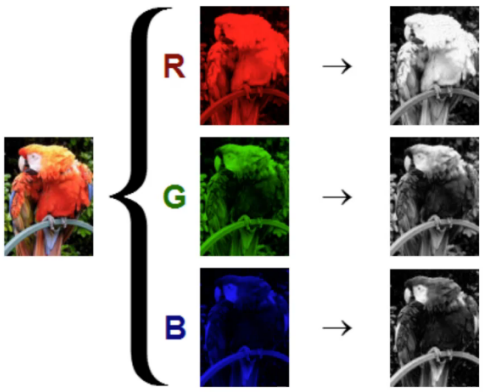
ERIAN DATA

Deep Learning





- But this was just in 2D for grayscale images.
- What about color images?
- Color Images can be thought of as 3D Tensors consisting of Red, Green, and Blue color channels.



- The shape of the color array then has 3 dimensions.
- Height
- Width
- Color Channels

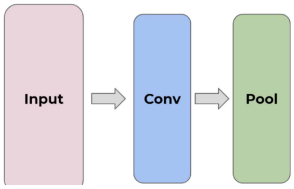
- This means when you read in an image and check its shape, it will look something like:
  - **(1280,720,3)**
  - **1280** pixel width
  - **720** pixel height
  - **3** color channels

- Often convolutional layers are fed into another convolutional layer.
- This allows the networks to discover patterns within patterns, usually with more complexity for later convolutional layers.

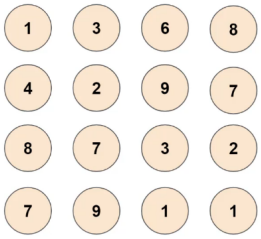
Pooling Layers:

- Even with local connectivity, when dealing with color images and possibly 10s or 100s of filters we will have a large amount of parameters.
- We can use pooling layers to reduce this.

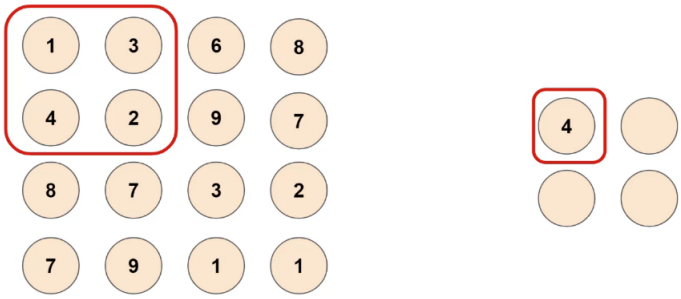
- Pooling layers accept convolutional layers as input:



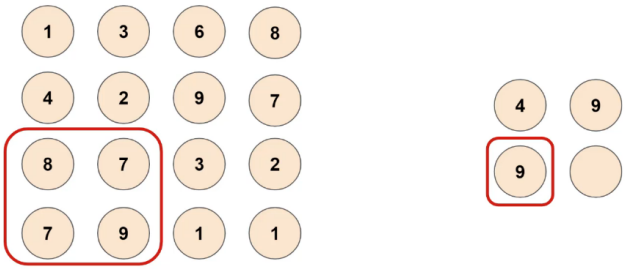
- Let's imagine a filter

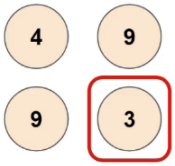


- Max Pooling: Window 2x2 , Stride: 2



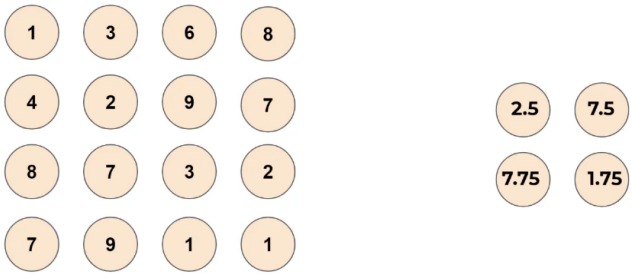
- Max Pooling: Window 2x2 , Stride: 2





Etc, etc until you end up with:

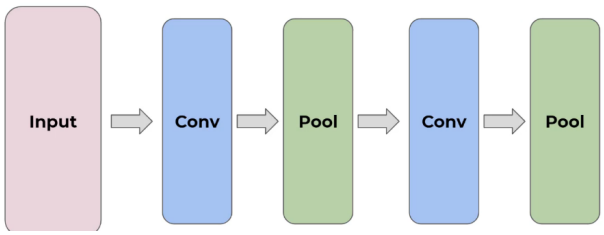
- Average Pooling: Window 2x2 , Stride: 2

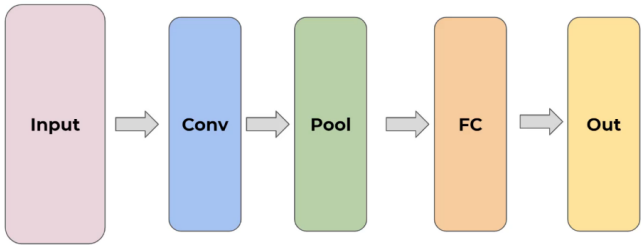


- This pooling layer will end up removing a lot of information, even a small pooling “kernel” of 2 by 2 with a stride of 2 will remove 75% of the input data.

- Another common technique deployed with CNN is called “Dropout”
- Dropout can be thought of as a form of regularization to help prevent overfitting.
- During training, units are randomly dropped, along with their connections.

- CNNs can have all types of architectures!





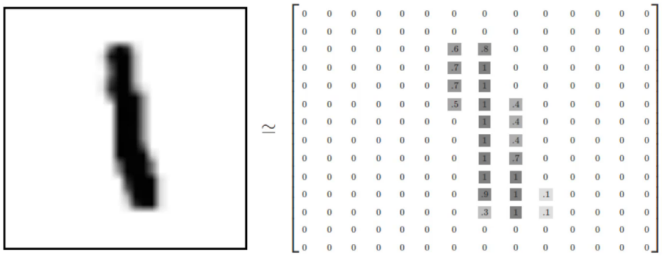
But they will all eventually lead to a FC and Out layer.

MNIST Data Set:

- 60,000 training images
- 10,000 test images

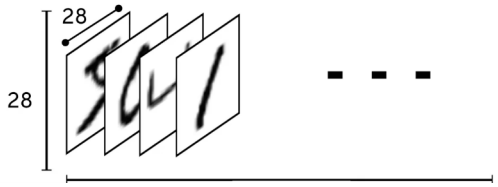
- The MNIST data set contains handwritten single digits from 0 to 9

- Specifically, 28 by 28 pixels



Digit is represented as a 2d array with a standardized value between 0 - 1.

- We can think of the entire group of the 60,000 images as a 4-dimensional array. 60,000 images of 1 chanel 28 by 28 pixels



- This array has **4** dimensions
  - (60000,28,28,1)
  - (Samples,x,y,channels)
  - For color images, the last dimension value would be 3 (RGB)

- For the labels we'll use One-Hot Encoding.
- This means that instead of having labels such as "One", "Two", etc... we'll have a single array for each image.

- The label is represented based off the index position in the label array.
- The corresponding label will be a 1 at the index location and zero everywhere else.
- For example, a drawn digit of 4 would have this label array:
  - [0,0,0,0,**1**,0,0,0,0,0]

```
model = Sequential()

model.add(Conv2D(filters=32,kernel_size=(4,4),input_shape=(28,28,1),activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(128,activation='relu'))

# OUTPUT LAYER SOFTMAX--> MULTI CLASS
model.add(Dense(10,activation='softmax'))

# keras.io/metrics

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

These are all hyperparameters based on your data. There is a correct value for them.

```
model = Sequential()

model.add(Conv2D(filters=32,kernel_size=(4,4),input_shape=(28,28,1),activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(128,activation='relu'))

# OUTPUT LAYER SOFTMAX--> MULTI CLASS
model.add(Dense(10,activation='softmax'))

# keras.io/metrics

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

These are hyperparmeters you can experiment with.

https://colab.research.google.com/drive/1H60qZQDQkrCDR2uD8S2MZdNb0kSTp2hK#scrollTo=q4rlrjEVdIU9

CNN on CIFAR-10:

https://colab.research.google.com/drive/1jU084BmtvaWHZmCYohec6dtKGdwI2LEL

Reminder:

1). X_train - This includes your all independent variables,these will be used to train the model, also as we have specified the `test_size = 0.4`, this means `60%` of observations from your complete data will be used to train/fit the model and rest `40%` will be used to test the model.

2). X_test - This is remaining `40%` portion of the independent variables from the data which will not be used in the training phase and will be used to make predictions to test the accuracy of the model.

3). y_train - This is your dependent variable which needs to be predicted by this model, this includes category labels against your independent variables, we need to specify our dependent variable while training/fitting the model.

4). y_test - This data has category labels for your test data, these labels will be used to test the accuracy between actual and predicted categories.

Real Data Sets:

https://colab.research.google.com/drive/1lZPvkF72emnK10VXGJZzJSnveBYNTxOF

Generating many manipulated images from a directory

In order to use .flow_from_directory, you must organize the images in sub-directories. This is an absolute requirement, otherwise the method won't work. The directories should only contain images of one class, so one folder per class of images.

Structure Needed:

- Image Data Folder
  - Class 1
    - 0.jpg
    - 1.jpg
    - ...
  - Class 2
    - 0.jpg
    - 1.jpg
    - ...
  - ...
  - Class n

Convolutional Neural Network Worksheet:

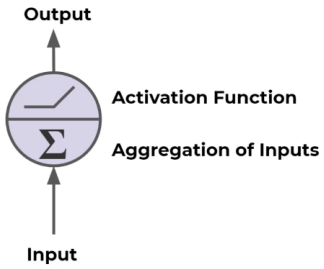
https://colab.research.google.com/drive/1qSAEOL4rTtGY6VLhEGcXc_V7pu-t7wnO#scrollTo=yR-5CGs5zbzK

Introduction to RNN's (Recurrent Neural Networks):

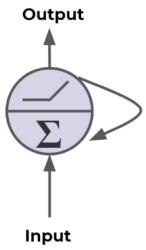
RNN's are more effective at sequence data.

- Examples of Sequences
  - Time Series Data (Sales)
  - Sentences
  - Audio
  - Car Trajectories
  - Music

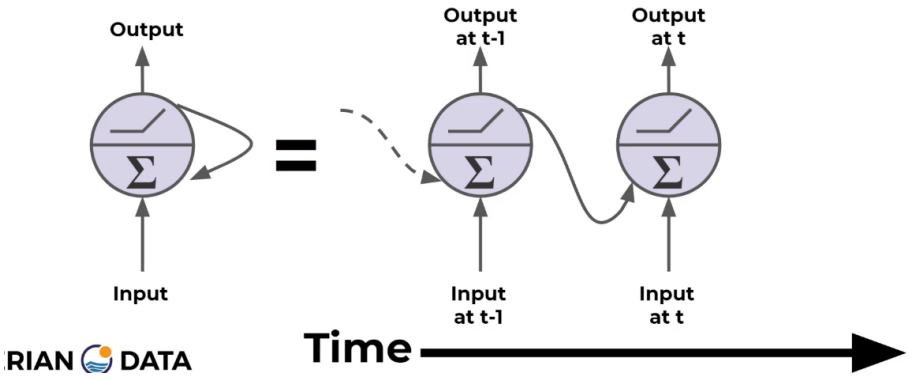
- Normal Neuron in Feed Forward Network



- Recurrent Neuron - Sends output back to itself!
  - Let's see what this looks like over time!



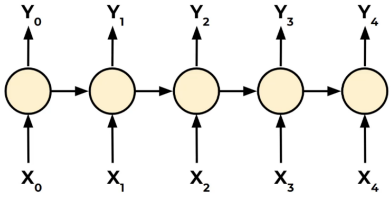
- Recurrent Neuron



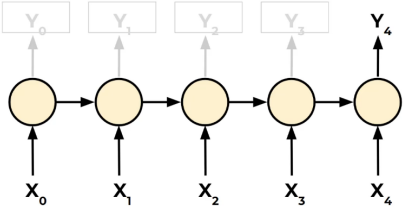
- Cells that are a function of inputs from previous time steps are also known as *memory cells*.
- RNN are also flexible in their inputs and outputs, for both sequences and single vector values.

Types of RNN's:

- Sequence to Sequence (Many to Many)



- Sequence to Vector (Many to One)



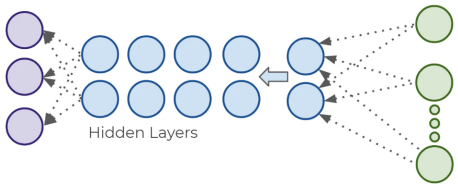
- A basic RNN has a major disadvantage, we only really “remember” the previous output.
- It would be great it we could keep track of longer history, not just short term history.

- Another issue that arises during training is the “vanishing gradient”.

Vanishing Gradients:

Exploding and Vanishing Gradient Units:

- Issues can arise during backpropagation

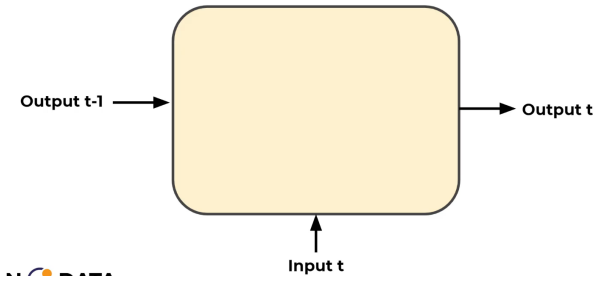


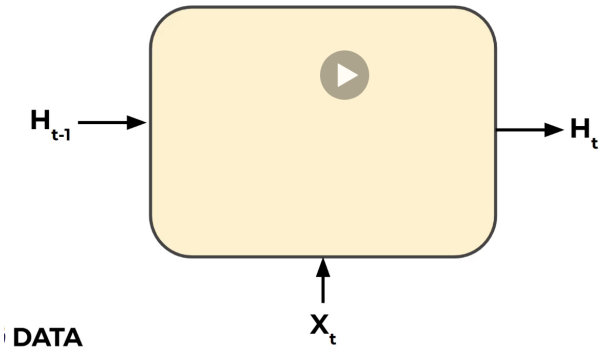
- Backpropagation goes backwards from the output to the input layer, propagating the error gradient.
- For deeper networks issues can arise from backpropagation, vanishing and exploding gradients!

- As you go back to the “lower” layers, gradients often get smaller, eventually causing weights to never change at lower levels.
- The opposite can also occur, gradients explode on the way back, causing issues.

LSTMS(Long Short-Term Memory) and GRU:

Normal RNN:

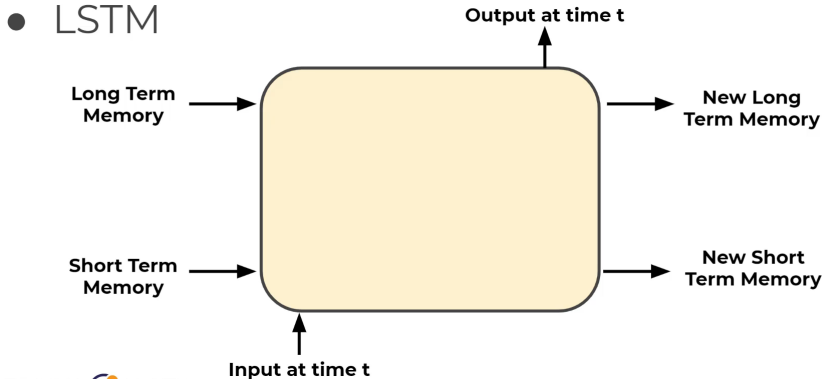


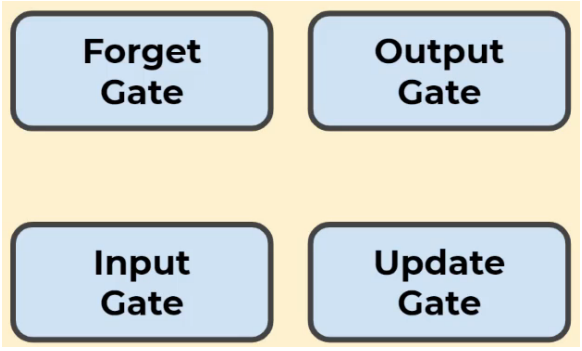


$$H_t = \tanh(W[H_{t-1}, X_t] + b)$$

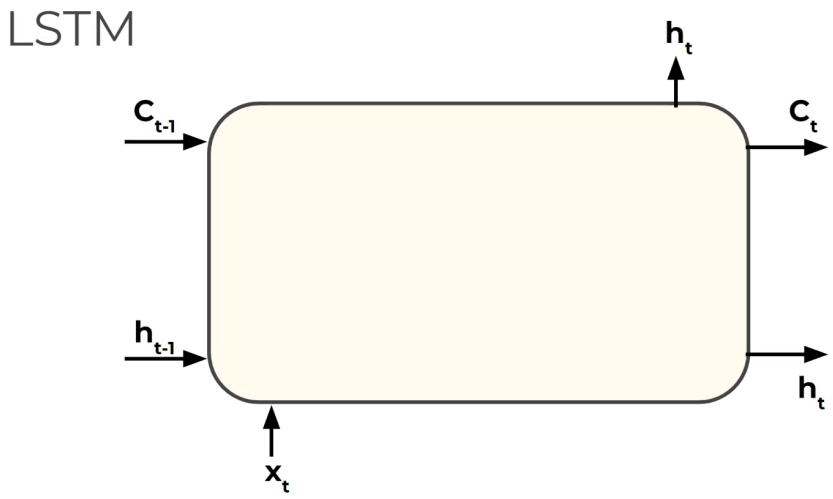
LSTM RNN:

- LSTM

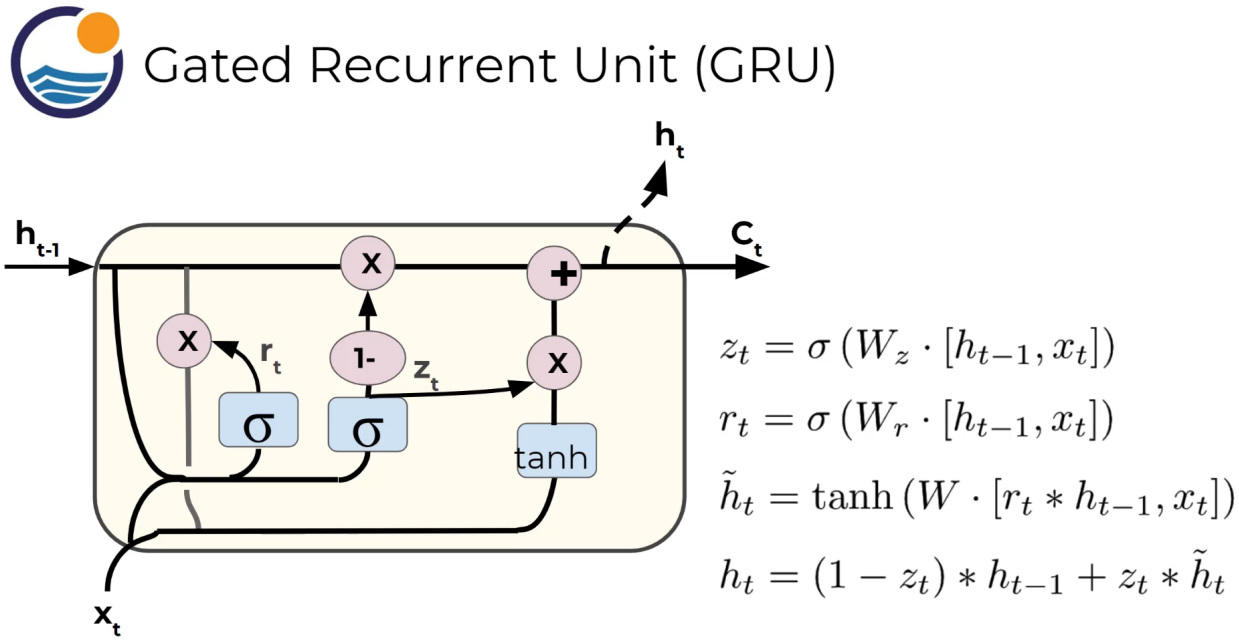




Gates optionally let information through.



One variation of LSTM is the GRU(Gated Recurrent Unit):



The further you predict the future with forecasted values. The more likely you are to get wildly varying future forecast values.

https://colab.research.google.com/drive/1qAhAJAQcjuVSJk323aLFv-kx8DAOeBB-#scrollTo=63Db_2WYyaiH

RNN on Time Series:

https://colab.research.google.com/drive/1HNS2E-j14Iq93dJPPGRGoiB3HhJW7hU8

RNN Worksheet:

https://colab.research.google.com/drive/1K1ZXSaEXFK7sbGMCOjNz8EbqBxPgMhPa

Natural Language Processing (Section 10):

- Step 1: Read in Text Data
  - We can use basic built in python commands to read in a corpus of text as string data.
  - Note, you should have a large data set for this, at least 1 million characters for realistic results.

- Step 2: Text Processing and Vectorization
  - The neural network can't take in raw strings, so we will encode them each to an integer.
    - A : 1
    - B : 2
    - C : 3
    - ? : 55

- Step 3: Creating Batches
  - We'll use Tensorflow's dataset object to easily create batches of text sequences.
    - ["h", "e" , "l" , "l" , "o" , " ", "m"]
    - [,"e" , "l" , "l" , "o" , " ", "m", "y"]

- Step 4: Creating the Model
  - We'll use 3 layers
    - Embedding
    - GRU
    - Dense

- Step 4: Creating the Model
  - Embedding Layer turns positive integers (indexes) into dense vectors of fixed size. eg. [[4], [20]] -> [[0.25, 0.1,0.3], [0.6, -0.2,0.9]]
  - Its up to the user to choose the number of embedding dimensions.

- Step 4: Creating the Model
  - GRU
    - Gated Recurrent Unit is a special type of recurrent neuron unit.
    - The GRU is like a long short-term memory (LSTM) with forget gate but has fewer parameters than LSTM, as it lacks an output gate.

- Step 4: Creating the Model
  - Dense Layer
    - One neuron per character.
    - Character labels will be one hot encoded so the final dense layer produces a probability per character.

- Step 5: Training the Model
  - We'll set up our batches and make sure to one-hot encode our character labels.

- Step 6: Generating new text
  - We'll save our models weights and show you how to reload a model's weights with a different batch size in order to pass in single examples.

https://colab.research.google.com/drive/1VB2kbs5yfZ1teywNND5L_5rfEzZLRD4k

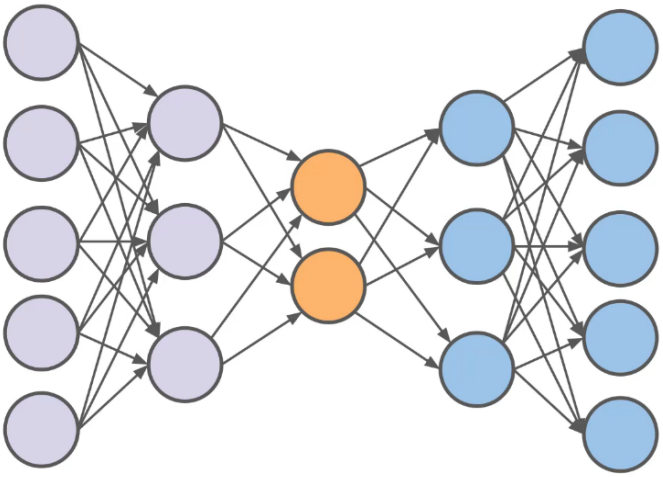
AutoEncoders (Section 11):

- Unsupervised learning means we won't have "correct" labels to compare our results to.
- Our use cases for autoencoders in this section are sometimes called semi-supervised.

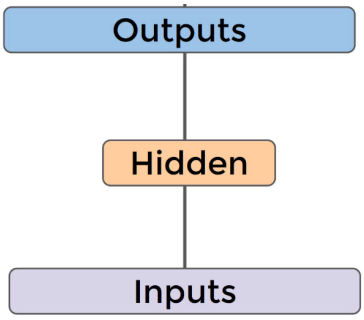
- The autoencoder is actually a very simple neural network and will feel similar to a multi-layer perceptron model.
- It is designed to reproduce its input at the output layer.

- The key difference between an autoencoder and a typical MLP network is that the number of input neurons is equal to the number of output neurons.
- Let's explore what this looks like and why we would use it!

Example Autoencoder



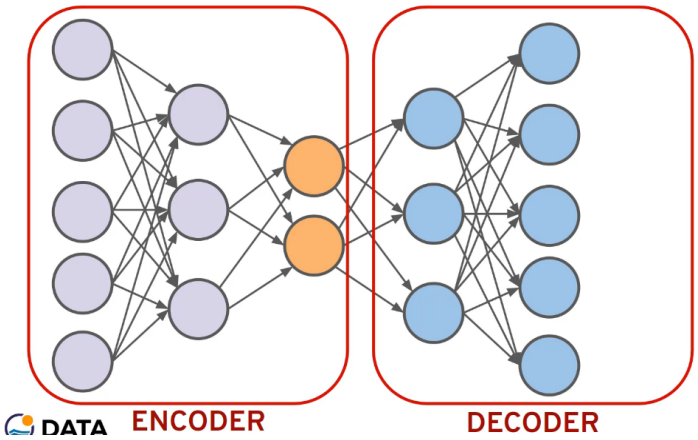
- This idea is extremely similar to PCA - Principal Component Analysis!



- ***Important Note!***
  - The hidden layer is **NOT** simply sub-selecting only certain features.
  - Its calculating combinations of the original features represent the original data in a reduced dimensional space.

- The main idea behind an autoencoder:
  - The center hidden layer reduced the dimensionality to learn the most important combinations of original features.

Example Autoencoder



AutoEncoder Practice:

https://colab.research.google.com/drive/1r_3dJ3Ez25QVXeWtgHRd9aecGvpOtlFW

AutoEncoder For Images:

https://colab.research.google.com/drive/1P6Ec6uVIZ5dL8XMaNW4DqThbbAPonDhi#scrollTo=8PA9SuGjD9ak

AutoEncoder Exercise:

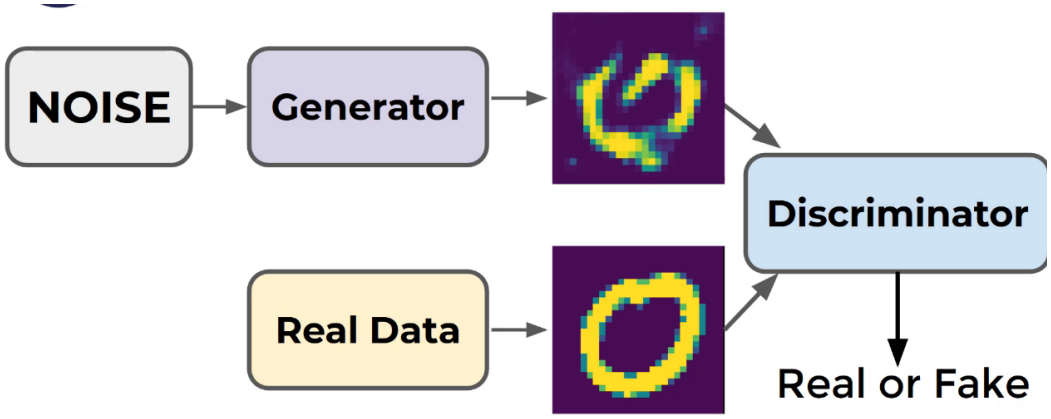
https://colab.research.google.com/drive/1mAVYCUE0mRZKftOb8J2n3vQK84_jqXmt

Generative Adversarial Networks (Section 12):

- GANs - Generative Adversarial Networks were invented in 2014 by Ian Goodfellow et al. and uses two networks competing against each other to generate data.
- GANs are often described as a counterfeiter versus a detective, let's see how they work.

- Generator
  - Recieves random noise (Gaussian distribution)
  - Outputs data (often an image)

- Discriminator
  - Takes a data set consisting of real images from the real data set and fake images from the generator.
  - Attempt to classify real vs fake images (always binary classification)



Two Phases: Train the Discriminator then Train the Generator.

- Phase 1: Discriminator
  - Real images (labeled 1) are combined with fake images from generator (labeled 0).
  - Discriminator trains to distinguish real from fake (with backpropagation only on discriminator weights)

- Phase 2: Generator
  - Produce fake images with generator.
  - Feed only these fake images to the generator with all labels set as real (1).
  - This causes the generator to attempt to produce images the discriminator believes to be real.

  - Because we feed in fake images all with labeled 1, we **only** perform backpropagation on the generator weights in this step.

https://colab.research.google.com/drive/1MQnLWe6zbPRDTOGxAemoYnDL7Kj9f7Eb

Bonus Deployment Section (Section 13):

https://colab.research.google.com/drive/1rGDuW2I_v8HQpw0GYnn1HkkuIN_E6jJD

Additional Things to Research:

- **capturing and analyzing network packet. Specifically for IoT devices.**
- **pcap files:**
- In other words, PCAP is the standard file used to capture network traffic for network performance analysis. Creating and analyzing a PCAP file is a basic component of your network analysis software.
- **IoT Sentinel**
- **The LSIF Dataset"**