**DEEP LEARNING**

* Çok fazla sayıda (milyon seviyesinde) parametresi vardır. Yine çok fazla sayıda hyperparametreleri vardır.
* Katmansal bir yapıdır. Katman sayısı ne kadar fazla ise o kadar deep seviyesi vardır.
* Bir katman varsa Neural Network, birden fazla sayıda katman (layer) varsa Deep Learning diyoruz. DL son yıllarda daha popüler olmuştur, çok sayıda katman kullanımına izin verdiği için.
* DL insan beyninin bir simülasyonudur.
* Andrew Ng, Jeff Dean öncülerdendir.
* DL de labeli olan datalar ile işlem yapacağız, o nedenle supervised learning kullanacağız daha çok, unsupervised kullanmayacağız. Unsupervised data ile de yapılabiliyor, ama çoğunlukla supervised data kullanılıyor.
* DL de data miktarı arttıkça performansı yani skorlar da artar. ML deki gibi overfitting riskine sebep olmuyor data fazlalığı.
* DL de biz resmi modele vereceğiz, future learning yapacak, arka işlem planıyla ilgilenmeyeceğiz. Biz predictionu doğru yapabilmesiyle ilgileneceğiz.
* DL neden yaygınlaştı?

Data miktarı arttı. Ayrıca günümüzde daha güçlü makineler mevcut. CPU/GPU ların gelişmiş olması. Yeni model ve algoritmaların olması.

* **Transfering learning** mevcut. Başkasının hazırlamış olduğu modeli biz alıp kullanabiliyoruz.
* DL de regularization için Lasso(L1), Ridge (L2) ile birlikte farklı yöntemler de göreceğiz.
* En popüler DL teknikleri:

- Multilayer Perceptron Networks(MNN)- Artifical Neuron Network (ANN) de denebilir. Daha önce yaptığımız projelerin sonuna bu model de eklenebilir.

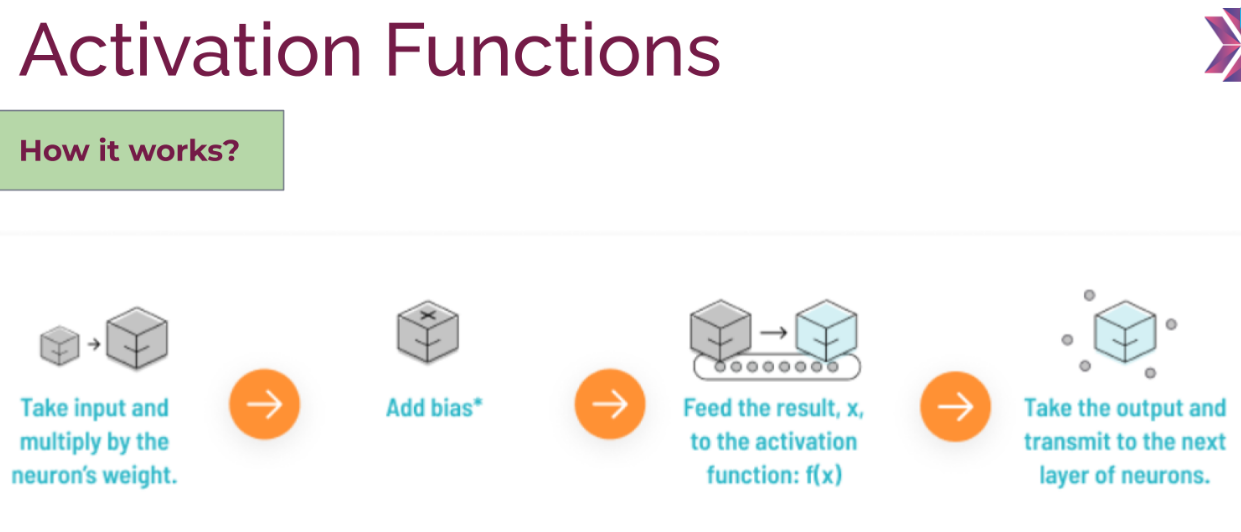
- Convolutional Neural Networks(CNN)-Data resimlerden oluşuyorsa kullanacağız.

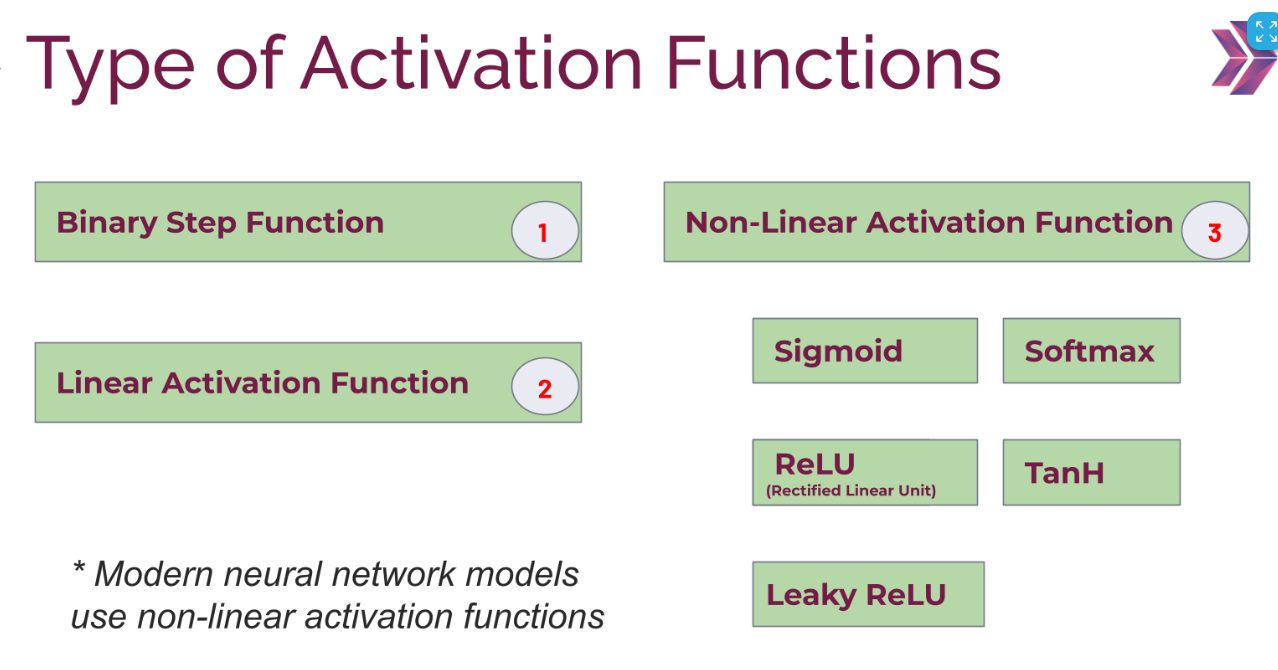
- Recurrent Neural Networks(RNN)-NLP alanında göreceğiz. Öncesi ve sonrası önemli olan (time seriesli- borsa gibi) datalarda kullanırız. Diğer örnek için cümledeki kelimelerin yer değiştirmesi anlamı değiştirmeyecek şekilde yapmamızı sağlar.

* DL, ML nin bir alt alanıdır.
* DL in transfer learninge izin veren bir yapısı vardır.

**Perceptron Models**

* Bias eklenecek, bu öğrenmeyi kolaylaştırmak için kullanılan bir ağırlıktır. Her bir nöronda tek bir tane bias var. Bias, lineer regresyondaki intercept ile aynı anlamda gibi.
* Bir nöron gelen ağırlıklı toplamları (inputları) alıp, ve bias ile toplayıp bir aktivasyon function ile output verir. Özetle bir nöronun 2 görevi vardır.
* Ağırlıklar 0 olduğunda outputun 0 çıkmasını engellemek için bias ekleyeceğiz.Bias eklemek bizim esnekliğimizi artırmış oluyor.
* Perceptron modeller XOR problemlerini çözemiyor, çünkü non-lineer problemi çözemiyor. Bu nedenle neural-networke geçmemiz gerekiyor.
* 1 neuron=1 node
* Bir nöronun outputu diğer nöronun inputu olabilmektedir.
* Activation functionu biz seçeceğiz.
* Ara katmanlara hidden layer, ilk katmana input layer, son katmana output layer denir. Hidden layer lar ne kadar çoksa o kadar deep bir model diyebiliriz. Laye riçindeki nöronlar ne kadar fazla ise o kadar wide denir.
* Input layer daki nöronlard activation function olmaz, asıl bir nöron gibi düüşünmemeliyiz.
* Her bir nöron kendinden sonraki gelen tüm nöronlara bağlıdır, buna “fully connected neural network” denir.
* Activation functions belirli bir aralığa sıkıştırır verileri (0 ile 1 aralığı gibi).
* Activation functions çok çeşitlidir, 15-20 civarı. **Biz en çok kullanılanları göreceğiz. Uygun dataya uygun activation function seçmek önemlidir.**
* Çok (50 civarı) katmanlı nöronlarda Swissh activation function kullanabiliriz.
* Activation functions computing maliyetini etkiler. Uygun seçim bu nedenle de önem arz etmektedir.

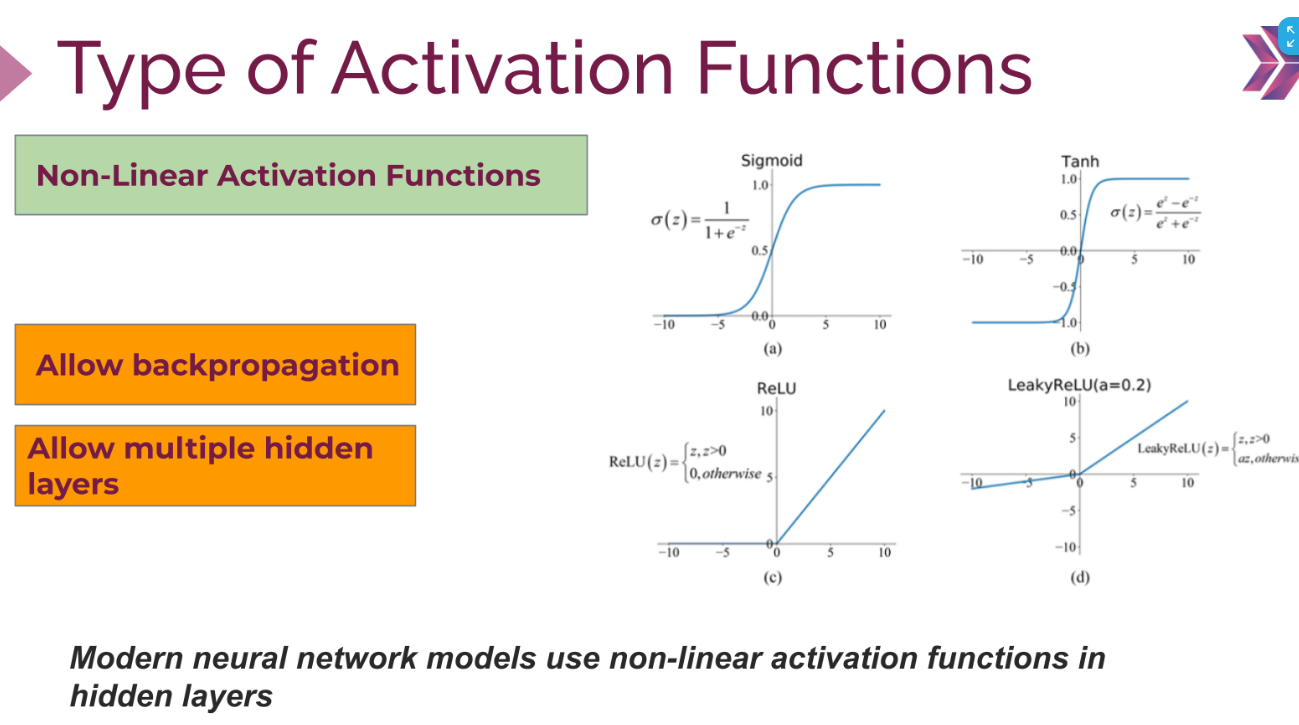




* **Binary Step Function:** Negatifleri 0, pozitifleri 1 yapar. Bunun yerine Sigmoid functionu kullanacağız.
* **Linear Activation Functions:** No Activation Function da denir. Kendisine gelen bilgiye bias ekleyip direkt iletir, ekstra bir functions işlemi uygulamaz. Hidden layerlar da bunu kullanamıyoruz. Türev alırken sorun oluşturacağı için, ağırlıkların güncellenmesine izin vermiyor, bundan dolayı. Non Linear Activation Functions tercih edeceğiz.

Modern neural networks uses Non Linear Activation Functions.

* **Non-Linear Activation Functions:**

****

* **Sigmoid functions:** Gradient Descent tabanlı çalışır arka planda, bize smooth gradient sağlayacak. Türev alınarak ağırlıklar güncellenecektir. 0 ile 1 arasında değer alır. S şeklindedir. Binnary classificationda kullanacağız.

Vanishing gradient olması, not zero centered olması hesaplama maliyetinin yüksek olması negatif özellikleridir.

* **TanH functions:** Data da olan negatif değerler bizim için önemli ise bu functionu tercih edebiliriz.
* 1e ve -1e yakın değerlerde eğimler çok düşük olması (vanishing gradient), hesaplama maliyetinin yüksek olması negatif özellikleridir.
* **ReLu functions:** Parçalı fonksiyonlu yapısı olduğundan işlevsel olabilmekte, özellikle hidden layerlar için. Hesaplama maliyeti daha azdır.

Negatif veya 0 değeri geldiğinde o veriyi türevi 0 olduğu için öldürür. Bu negatif yönüdür.

* **Leaky ReLu functions:** ReLu nun negatif yönünün giderilmesi için üretilmişir, negatif değerleri az da olsa ağırlıklandırır.

Sonuçların tutarsızlığından kaynaklı bu çok kullanılmamaktadır. Biz en çok ReLu yu kullanacağız.

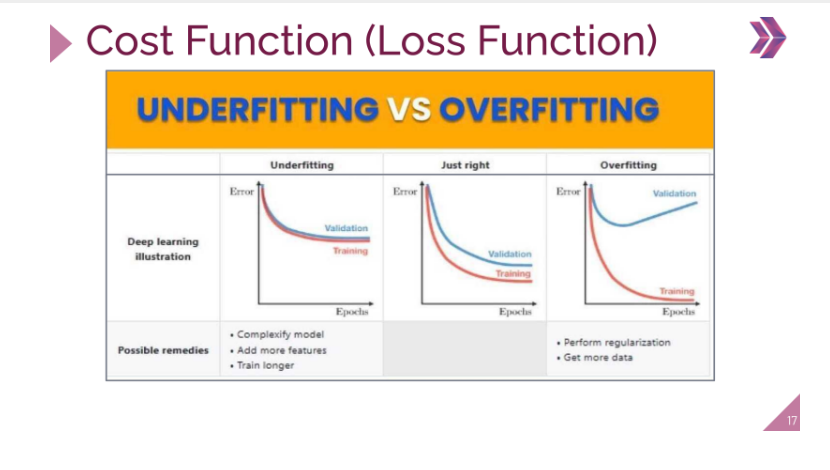
* **Softmax functions:** Multiclasslı output için daha kullanışlıdır. Toplamları 1 olacak şekilde 0 ile 1 arasında değerler verir.

**Notlar:**

* Her katmanda farklı function seçilebilir.
* Binary classification🡪Sigmoid functions
* Multiclass classification🡪 Mutually Exclusive Classes 🡪Softmax functions

🡪 Non-Exclusive Classes 🡪Sigmoid functions

* **Cost (Loss) function:** Modelin ne miktarda yanlış olduğunu, hatayı ölçer. Coss function bir gözlem için olan hata, loss function ise tüm gözlemlerin hatasının ortalamasıdır. Coss function ne kadar küçük ise o kadar makbuldür. Bu nedenle coss functionu minimize etmeye çalışacağız. En başta weights ve bias random şekilde atanır, biz sonra ağırlıklarını güncelleyeceğiz. Local optima seviyesine çekmeye çalışacağız.

****

**Train longer🡪 epoch sayısını artırmak**

**Complexity model🡪hidden layer sayısını artırmak**

**Add more features 🡪 daha fazla feature eklemek**

* **Types of Cost functions:** 
  + **Regression Loss Functions**

model.compile(optimizer=’adam’, loss=’mse’) 🡪 kodunu kullanacağız

Burada mse yi dikkate alarak ağırlıkları güncelle demiş oluyoruz.

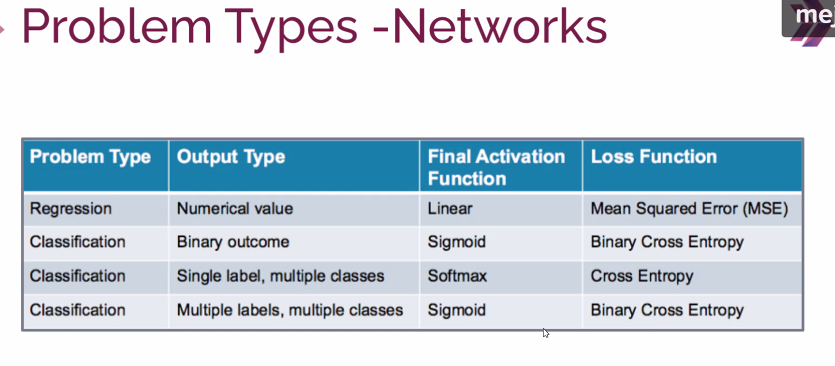
* + **Binary Classification Loss Functions**

model.compile(optimizer=’adam’, loss=binary-crossentropy) 🡪 kodunu kullanacağız

* + **Multi-Class Classification Loss Functions**

model.compile(optimizer=’adam’, loss=categorical\_crossentropy) 🡪 kodunu kullanacağız

* Datayı Train, Validation, Test şeklinde split edeceğiz.
* Train ve Validation karşılaştırarak underfiting/overfitting durumunu inceleyeceğiz.



**11 mayıs 2023**

* **Gradient Descent:** Loss functionu optimize etme algoritmasıdır.

**Epoch:**Train datasının modele verilmesinin sayısıdır. Epoch kaç ise data o kadar kez modele sokulmasıdır.

**Batch size:** modelin 1 defa modele girerken, datanın kaç adet datadan oluşacağını verir**.** Aynı anda modele beslenen gözlem sayısı da diyebiliriz.

* Makine öğrenmesinde, bir epoch, eğitim sırasında tüm veri kümesinin tam bir döngüsünü ifade eder. Bir epoch sırasında, model her bir eğitim örneğiyle karşılaşır ve parametrelerini verilere daha iyi uyacak şekilde günceller.  
  Öte yandan, batch size ise her epoch sırasında modele aynı anda beslenen eğitim örneklerinin sayısını ifade eder. Örneğin, batch size 10 ise ve veri kümesinde 200 örnek varsa, model her epoch sırasında 20 kez parametrelerini günceller (iterasyon) ve her güncelleme 10 örneğe dayanır.  
  Özetlemek gerekirse, bir epoch tüm veri kümesinin tam bir döngüsünü ifade ederken, batch size her epoch sırasında modelin işlediği örnek sayısını ifade eder.

model.compile(optimizer=’adam’, loss=’mse’) de **optimizer bölümüne gradient descent türlerinden birini yazacağız.**

* **Step size = slope \* learning rate (learning rate = adım büyüklüğü)**

**Türev düştükçe adım büyüklüğü azalmış olur.**

* Derin öğrenmede iyi performans elde etmek için uygun bir öğrenme oranı (learning rate) seçmek önemlidir. Genellikle, çok düşük bir öğrenme oranı yavaş yakınsama ile sonuçlanırken, çok yüksek bir öğrenme oranı kararsızlık ve kötü yakınsama neden olabilir.
* ML de cost functionlar Convex yapıya sahiptirler. DL'da ise yapısı gereği convex  bir yapı sağlanamaz. Bu sebepten Local ve global minimumlar oluşur.
* Eğim pozitif ise sola, eğim negatif ise sağa gidecektir.
* **Gradient Descent (variants-optimizer)**

**Batch (Vanilla) GD:** Model 1 epoch ile update olur. Bu hafızayı yoran bir şeydir, tüm datayı işleme alır, hepsinin sonucunu hafızada tutması gerekir.

**Stochastic GD:** Her bir example için iterasyon yapılır**.**

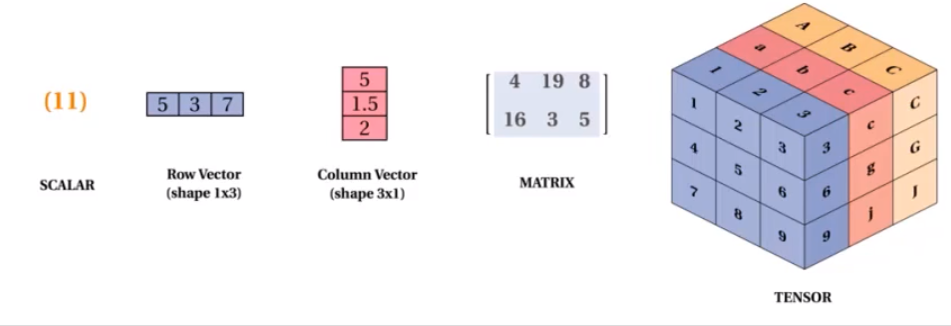
**Mini-Batch GD:**  İlk ikisinin ortası bir durumdur. Belirtilen batchlere uygun ayırıp iterasyon eder. Türevler alarak geriye doğru günceller.

**Adadelta:**

**Adam:** Biz daha çok bunu kullanacağız.

**AdamRMSProp**: Bazen de bunu kullanacağız.

* Hangi optimizeri ne zaman kullanacağımız datadan dataya değişir. Deneyerek anlayacağız.
* **Backpropagation (geri yayılım): Gradient descentin türev almasını sağlayan şeydir. Her iterasyonda ağırlıklar güncelenerek devam edecektir.**
* Backpropagation algoritması, temel olarak iki aşamadan oluşur: İleri yayılım (forward propagation) ve geri yayılım (backward propagation). İleri yayılım aşamasında, girdi verileri sinir ağından geçer, ardışık katmanlarda işlenir ve çıktı üretilir. Çıktı ile gerçek değer arasındaki farka göre bir hata ölçüsü hesaplanır.
* Geri yayılım aşamasında, hatayı geriye doğru yayarak her bir ağırlık değeri ve bias için hata türevleri hesaplanır. Bu türevler, hatayı azaltmaya yönelik bir güncelleme yapmak için kullanılır. Geri yayılım, zincir kuralı (chain rule) yardımıyla gerçekleştirilir. Hata türevleri, katmanlardan geriye doğru hesaplanır ve ağırlık değerleri bu türevler kullanılarak güncellenir.
* Bu süreç, verilerin ağ üzerinde birçok kez ileri ve geri geçtiği tekrarlı bir şekilde gerçekleştirilir. Backpropagation algoritması, genellikle bir optimizasyon algoritması olan gradyan inişi (gradient descent) ile birlikte kullanılır. Gradyan inişi, hata azaltmak için ağırlık değerlerini güncellerken backpropagation algoritmasından elde edilen türevleri kullanır.
* Backpropagation, derin öğrenme (deep learning) ve sinir ağlarındaki öğrenme süreçlerinin temel bir bileşenidir. Bu algoritma, sinir ağlarının eğitilmesinde önemli bir rol oynar ve ağırlık değerlerinin doğru şekilde ayarlanmasını sağlar.
* **DL Kütüphaneleri**
* DL de **Keras** ve **Tensorflow** kütüphanelerini kullanacağız. Birçok kütüphane var, **PyTorch** da bunlardan biridir. Tensorflow ı okuması daha kolay. Tensorflowun farklı servisleri var, daha esnek.Birlikte kullanılan Keras ile Tensorflow un daha user friendly olmasını sağlıyor.Tensorflow opensource bir kaynaktır. Kendisine özgü görselleştirme kiti mevcuttur.
* ML modelelrini tensorflow ile de eğitebilir. Bu da bir artısıdır.
* Ayrıca birçok ML modelini Tensorflow kullanarak da eğitebiliyoruz.
* **Tensor**: lineer cebir yapılı (vektörler ve matrisler gibi) bir data türüdür. Tensorlar arraylara göre çok dahahızlı çalışır.



Tensorflow install ederken hata veriyorsa anaconda promptta şu kod yazılmalıdır: conda install tensorflow

* DL algoritmaları multicollinearity ile mücadele edebiliyor, multicollinearity olması soruns oluşturmuyor.
* DL bizden datayı tensor veta array olarak istiyor. Scale yaptığımız anda bu formata dönüştüğü için sorun olmuyor.
* Data leakage olmaması için önce train-test split sonra scaling yapmalıyız.
* Minmax scaler datayı 0-1 gibi küçük bir aralığa soktuğu için genelde bunu tercih edeceğiz. Ancak istenirse diğer scaling türleri de seçilebilir.
* Output layerin özel bir ismi yoktur. Denseleri import ederken input ve dense import etmek yeterlidir.
* Notebook (price tahmini) olduğu için regresyon problemidir. Bunlarda en sona activation function adı yazmamıza gerek yoktur.
* Parametre sayısı nöron sayısının çarpımlarına bias eklenerek bulunur.
* Learning rate optimizerin içine girip ordaki bir parametredir.
* Modeli kaydederken save.model diyerek h5 uzantılı şekilde kaydediyor. Biz bu modeli sonra alıp sonra tekrar kullanabiliyoruz.
* Early stopping, normalleştirme tekniklerinden biridir, öyle ki, validation setindeki validation gelişmiyorsa, tüm çağlardan geçmek yerine dururuz. Ayrıca ağdaki rastgele düğümleri devre dışı bırakabilir ve tek bir modelin eğitim sırasında her seferinde farklı bir ağ gibi davranmasını sağlayabiliriz. Bu tekniğe dropout denir.
* Restore\_best\_weights = True seçilmeli . Güncellediği ağırlıkları kullanmasını istediğimiz için.

Callbacks = [early\_stop] (liste içinde yazılmalı)

* Regularization teknikleri: early stopping , dropout.

Early stopping: patience değeri ile kontrol aralığı veriyoruz. Fit işleminde loss func azaldı ve artmaya başladı ise Patience sayısı kadar epoch boyunca bakıyor ve artma devam ediyorsa burda durduruyor fit işlemini.

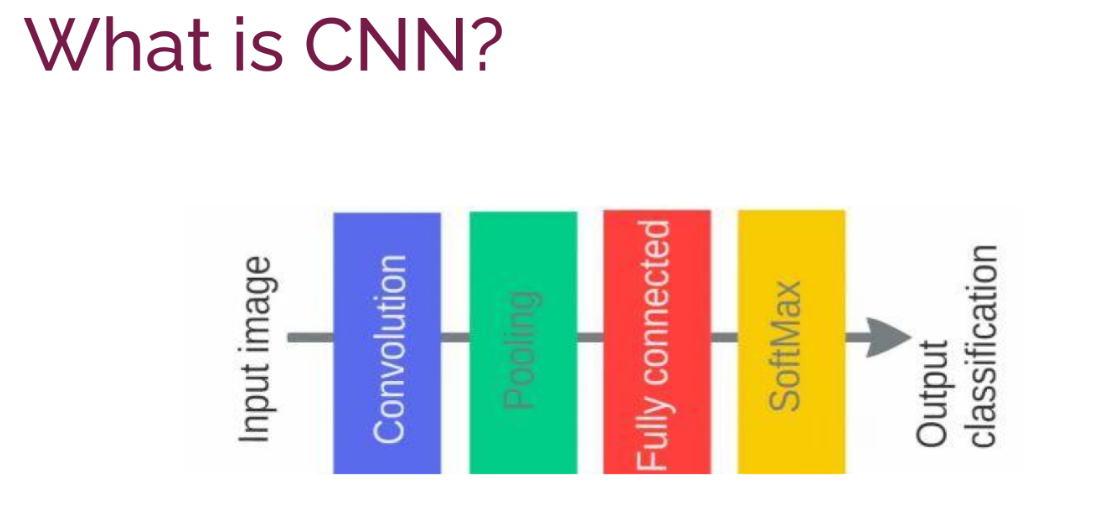
Dropout: bazı nöronların kapatılıp onlara bilgi girişinin kapanması şeklindedir. Datayı ezberlememesi için çomak sokuyoruz, gibi düşün 😊

* DL demodeli kaydetmek için pickle yerine kendine has metotları kullanacağız. model.save ile ht5 formatında kaydedeceğiz.
* Transfer learning DL de mümkün. Çünkü modeli kaydettiğimizde model architecti vb içinde barındırıyor.
* DL bir unstructured datalarda(resim gibi), ikinci olarak da çok büyük datalarda kullanmak elverişli olacaktır. Diğer durumlarda ML modelleri tercih edilebilir.
* Overfitting durumunda epoch sayısınıartırıp, early stop kullanmak best practise olabilir.

**17 Mayıs**

**CNN**

* CNN supervised datalarda kullanılmaktadır.
* CNN face detection, face verification, face classification, facial emotion recognitionda kullanılır.
* Object detection (yaya, trafik lambası olanları seç), self-driving cars, cancer detection, image captioning (resimde olanı anlatıyor), midjourney (cümle ile ilgili resi verir), OCR (Optical Character Recog), handwritten character recognition.



* 2B veya 3B fotoğrafları katman katman öğrenecek.
* İnput feature extraction + classification ile öğrenip pooling (resmin ebatıyla ilgili) ve devamında flattening ile ANN in kullanabileceği formata çevirecek. En son output verecek.
* CNN mimarisi 2 tane bileşenden oluşuyor.

1. Fotoğraflar üzerinden feature learning ve

2. Classification part.

* Flattening yapmak demek tüm bileşenleri çarpıp tek bir vektör elde etmek demektir. Örneğin, 64\*64 ebatlı 3 renkli bir resim ise 64\*64\*3 = 12.228 feature üretmiş oldu. Modele resmin kaç kanallı olduğunu söylememiz gerekiyor. Çünkü kullanacağı filtreyi buna göre kendisi ayarlıyor. Renkli resimler RGB olanlar, 3 kanallı olarak ifade edilir.
* Sistemin imputa verdiği cevaba **convolution (evrişim)** denir. **Convolution filtreler** bulunur, filtre ebatını biz ayarlarız (hyperparametre ile). 3\*3, 5\*5, …, 11\*11 gibi olacak.
* **Convolution işlemi** image datalardaki resmin üzerinde filtre gezdirerek, feature learning yapmamızı sağlar. **Filtre** tutulur, etraftaki pikselleri çarpıp toplayıp merkeze yazar.
* Convolution işlemi yapılırken kenarlar ve köşelerdeki pikseller kaybedilir. Bunu engellemek için **padding** işlemi yapılmalıdır.
* Filtreler 2 ye ayrılır: Hazır filtreler ve learnable(öğrenebilen) filtreler. Hazır filtreler CNN öncesinde kullanılıyordu. CNN in en büyük farkı learnable(öğrenebilen) filtre kullanabiliyor olmamızdır.
* Filtre türleri kaydırma yönü sayısına göre 1D, 2D, 3D formatta olabilir. Biz daha çok 2D olarak kullanacağız.
* Hazır filtre örnekleri: edge detection 🡪 kenar belirlemeye yarar. Colour detection🡪 renk tonu belirler. 1 lerin olduğu hizaya önem verir.
* **Stride**: Filtreyi kaydırma sayısıdır. 1 olursa filtreyi 1er 1er kaydırıyor. Hem x ekseni hem y ekseni boyunca. Stride arttıkça kenar ve köşelerden kaybedilen piksel sayısı artar. Resimde obje etrafı (foto kenar ve köşeleri) önemsiz olduğunda stride büyük tercih edilebilir.
* Padding (doldurma) kenar ve köşeleri kaybetmek istemiyorsak(same padding) kullanırız. İnput ile output aynı boyutta olur. İkinci olarak da, elimizdeki fotoların boyutunu ayarlamada (valid padding) kullanırız.
* Valid padding kısmında bunları uygun şekilde seçeceğiz.
* Ebat için 5 çıkarıp 1 ekliyoruz. Kanal sayısı ile çarpıyoruz. 32\*32 için ebat 27\*27 oldu.
* **Pooling**: Kaydırarak poollar oluşturur, boyut azalmış olur. Pooling de default stride 2\*2 dir. İki türü vardır. Pooling yapma tercihe bağlıdır. Çalışma maliyetini azaltır pooling yapmak. O nedenle tercih edilebilir.
* **Max pooling**: Kaydırma yaparak her pooldaki maksimum değeri alıyor. Daha çok bunu tercih ederiz.
* **Average pooling**: Filtre gibi kaydırma yaparak her pooldaki değerlerin ortalamasını alıyor.
* **Flattened** işlemini ANN e sokabilmek için yapıyoruz. Fully connected yapıya dönüştürüyoruz böylece.
* Datadaki resimlerin ebatları faklı ise EDA yaparak ebatları aynı hale getireceğiz. (compute vision kütüphanesi ile)
* Colab Notebookta 🡪Runtime dan GPU seçip kaydedidilir.
* Piksellerde 0 : siyahı, 255: bembeyazı temsil eder.
* Class sayımız kadar nöron olması gerekiyor. Mesela 10 classlı bir data ile çalışıyor isek, 10 nöron oluşturacağız.
* Activation fonksiyonu olarak softmax kullanıyoruz. (classification problemi olduğu için). Softmax bize olasılıklar döndürüyor, tüm olasılıkların toplamı 1 oluyor. Burda 10 nöronun toplamı 1 olacak.
* One hot encoder/dummies yerine to\_categorical kullanacağız.
* Gradient descent sayesinde filtreler öğrenmiş oluyor.
* CNN de bir diğer regularization tekniği (overfitting olmasını engeleyen) BatchNormalization dur. Bu teknik ANN de kullanılabilir. Ancak CNN de daha iyi sonuçlar verdiği için CNN de kullanmayı tercih edeceğiz.
* BatchNormalization nin 4 parametresi vardır. 2 si öğrenebilen parametredir, sürekli güncelenecekbu parametreler. Bunlar: gama, beta. Diğer ikisi mean ve standart sapmadır. Bunlar normal parametrelerdir.
* BatchNormalization daha smooth bir iniş sağlar. EKG gibi grafik çıkmasını engeller.
* Diğer regularization teknikleri: Early Stopping, Dropout, L1L2. Şimdi BatchNormalization eklendi bu tekniklere.

**20 Mayıs**

* **Image Data Generator** kullanarak fotoğrafları transform ediyoruz, modele bu transform edilmiş formları veriyoruz. İmage data generator, döndürme, simetri alma, parlaklık verme, … gibi çok fazla opsiyonu var. Bunu kullandığımızda örneğin yüz tarama üzerinden düşünürsek yüzün farklı yön ve açılardan görüntüsünü de tanımasını sağlar. Ayrıca overfittinge gitmesini engellemiş olur.
* Image data generator kullanıyorsak batch size ı fit içerisinde değil, dışarıda belirliyoruz(GPU yu yormaması için).
* Classlar imbalanced olursa;

1. Önce skorlara bakacağız. Model öğrendi mi, bunu anlayacağız.
2. Model öğrenememiş ise, class weight ile oynadık, optun ile. Optuna da recal sınıfının değerlerinin maximize etmeye bakacağız.
3. … 7. Dersin donlarında cevapladı hoca.

**22 Mayıs**

* Resimler 1, 3 veya 10-11 chanel lı da olabiliyor. Uydudan çekilen fotoğraflarda 10-11 chanel lı kullanılabiliyor.
* # overfitting
* - early stop
* - drop out
* - reduce patience (especially CNN)
* - learning rate
* - batch normalization
* - Image data generator for CNN (3.3) (data augmentation)
* - İmage size çok büyük secilirse (örnek 300x300) overfttşnge goturebilir
* \* Imbalanced dataset:
* - class weight (sklearn): first check&compare scores for each classes. If scores points to an imbalanced data, then increase the weight of the class with lower scores
* - make\_scorer for multiclass
* - Update optuna to max the recall of the class with lower scores
* - y\_pred = pred\_prob > 50 yerine 0.35 yapabiliriz ornegin.threshold
* Ama bunlari yaparken recall ve precision dengesini de dikkate almak gerekir. Datanin hassasiyetine göre kullanacagimiz metric ve/veya yöntem de değisecektir. Cnn ve Ann'deki yontemler ayni
* Restore\_best\_weights=True🡪Early stopping olanyere göre modeli şekillendir demek.

image datalarda niçin ANN yerine CNN tercih ediyoruz?

CNN'ler (Convolutional Neural Networks), görüntü verilerinin işlenmesi ve analizi için özellikle tercih edilen yapay sinir ağlarıdır. İşte image datalarda neden CNN'lerin ANN'lere (Artificial Neural Networks) tercih edildiğini anlatan bazı nedenler:

1. Bölgesel Bağımlılık: Görüntü verilerinde, pikseller arasında bölgesel bağımlılıklar bulunur. Yani, bir pikselin etrafındaki pikseller, onun değerini ve anlamını etkiler. CNN'ler, bu bölgesel bağımlılıkları yakalamak için özel olarak tasarlanmışlardır. Evrişim (convolution) katmanları, pikseller arasındaki ilişkileri modelleyerek görüntü verilerindeki önemli özellikleri çıkarır.
2. Paylaşılan Ağırlıklar: Görüntüler genellikle büyük boyutlara sahiptir ve yüksek boyutlarda işlem yapmak kaynak yoğunluğuna neden olabilir. CNN'ler, evrişim katmanlarında paylaşılan ağırlıklar kullanarak parametre sayısını azaltır. Bu, modelin daha hızlı öğrenmesini sağlar ve aynı zamanda daha az bellek ve işlem gücü gerektirir.
3. Ölçeklendirme: CNN'ler, ölçeklendirilebilir bir yapıya sahiptir. Farklı boyutlardaki görüntülerle çalışabilirler ve giriş boyutlarındaki değişikliklere uyum sağlarlar. Bu özelliği sayesinde, farklı çözünürlüklerdeki görüntüler üzerinde etkili bir şekilde çalışabilirler.
4. Özellik Çıkarımı: CNN'ler, evrişim ve havuzlama (pooling) katmanları aracılığıyla önemli özellikleri otomatik olarak çıkarabilirler. Bu sayede, giriş verilerinin daha düşük boyutlu bir temsilini elde ederler ve bu temsil üzerinden sınıflandırma, nesne tanıma, yüz tanıma gibi görevleri gerçekleştirebilirler.
5. Görsel Veri İşleme Uygunluğu: CNN'ler, görüntü verileriyle daha uyumlu bir şekilde çalışır. Evrişim ve havuzlama katmanları, özellikle görüntü verilerinde bulunan lokalite, translasyon invariyansı gibi özellikleri yakalamak için tasarlanmıştır. Bu da görüntülerdeki yapısal bilgiyi daha iyi bir şekilde kullanmalarını sağlar.

Sonuç olarak, CNN'ler özellikle görüntü işleme, görüntü sınıflandırma, nesne tanıma gibi görevlerde başarılı sonuçlar veren ve görüntü verilerinin özelliklerini daha iyi yakalayabilen bir

CNN de neden RELU aktivasyon fonksiyonu kullanırız?

CNN'lerde ReLU (Rectified Linear Unit) aktivasyon fonksiyonu sıklıkla tercih edilir ve bunun birkaç nedeni vardır:

1. Etkili Derin Öğrenme: ReLU, derin sinir ağlarının eğitimini hızlandıran ve daha etkili hale getiren bir aktivasyon fonksiyonudur. Bu fonksiyon, geri yayılım algoritmasının etkin bir şekilde çalışmasını sağlar ve gradyanın daha iyi yayılmasına yardımcı olur. ReLU'nun doğrusal ve basit yapısı, hesaplama maliyetini de azaltır.
2. Sıfır Merkezli Dağılım: ReLU, negatif girişlerde sıfır çıktısı üretirken, pozitif girişleri de doğrudan iletmektedir. Bu özelliği sayesinde, ReLU aktivasyon fonksiyonu nöronların çıktılarının sıfıra yakın ve dağılımın sıfır merkezli olmasını sağlar. Bu durum, ağın daha fazla öğrenme kapasitesine sahip olmasını ve aşırı uyum (overfitting) riskini azaltmasını sağlar.
3. Düşük Gradyan Vanishing Etkisi: ReLU, bazı diğer aktivasyon fonksiyonlarına göre gradyan vanishing (sıfıra yakınsama) sorununu daha az yaşar. Özellikle sigmoid veya tanh gibi doygunlaşan aktivasyon fonksiyonlarında, gradyanın küçük değerler alması nedeniyle derin ağlarda geriye doğru gradyanın kaybolması sorunu ortaya çıkabilir. ReLU, pozitif girişlerde gradyanı doğrudan ilettiği için bu sorunu azaltır.
4. Hesaplama Verimliliği: ReLU'nun hesaplama maliyeti oldukça düşüktür. Doğrusal bir fonksiyon olduğu için, diğer aktivasyon fonksiyonlarına kıyasla hesaplama süresi daha kısadır. Bu özellik, büyük ve karmaşık CNN'lerin daha hızlı çalışmasına katkıda bulunur.

Ancak, ReLU aktivasyon fonksiyonunun bazı dezavantajları da vardır. Özellikle negatif girişlerde sıfır çıktı üretmesi, bazı durumlarda "ölü nöron" problemine yol açabilir. Bununla birlikte, bu dezavantajlarını telafi etmek için ReLU türevleri olan Leaky ReLU, Parametric ReLU ve ELU gibi varyasyonlar geliştirilmiştir.