Deep Learning Tutorial

Prepared by Enes Okur. Feel free to use for educational purposes.

Concepts in Deep Learning

How Does Deep Learning work?

ŷ = $\sigma(x1 * w1 + x2 * w2 + b)$

$\sigma(x)$ = $\dfrac{1}{1+e^{-x}}$

This formula is for a single node. Imagine we have two variables input to the node. Those two variables are multiplied with variables called weights and then a term called bias is added. This results a linear prediction. But we want a prediction for nonlinear problems too. For that reason, we apply nonlinear functions to the result and we get a new result. Those functions are called activation functions. Sigmoid function is applied in the output layer for turning result into some kind of probability. It is used for binary classification problems. (Weights and biases are parameters learned by the model during a process called backpropagation. Initially, they can be set any random value. Model will adjust itself at the end of the backpropagation process. Weights determine how important the variable is.)

Forward Propagation: Calculating output of nodes in the model towards output.
Backward Propagation: Changing weights and biases according to how wrong the model predicted.

Dummy Encoding

Dummy variables are numerical variables that represent the actual data. For Instance, given male and female gender, you could give 0 to represent the males and 1 for the females. This gives an actual representation of the information and and the information is in numerical format which can be integrated into the machine learning model.(Creates a single vector.)

One hot Encoding

We map each category to a vector that contains 1 and 0 denoting the presence of the feature or not. The number of vectors depends on the categories which we want to keep.

Transpose

The transpose of a matrix is obtained by interchanging rows into columns or columns into rows.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}^{\mathsf{T}}$$

Creating Maxtrix

np.array() in numpy library.
torch.tensor() in pytorch library.

Example:
nV = np.array([[1,2,3,4,5]]) // or torch.tensor([[1,2,3,4,5]])
nVT = nV.T // nVT is the transpose of nV

Dot Product

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} . \begin{bmatrix} d \\ e \\ f \end{bmatrix}$$

ad + be + cf = |a| * |b| *cos(α)
α = angle between vectors. normalized dot product shows how similar two vectors are.

$$\frac{ad + be + cf}{|a| * |b|} = \cos(\alpha)$$

if cos value is closer to 0 that means angle is closer 90(vectors are very different.)
if cos value is closer to 1 that means angle is closer 0(vectors are similar.)

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} . \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aa + bb + cc + dd + ee + ff + gg + hh + ii$$

np.dot() or tensor.dot()

Matrix Multiplication

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

np.matmul() or @ sign.
matrix multiplication is only valid when axb * bxd = axd

Softmax Function

$$f(xi) = \frac{e^{xi}}{\sum_{j=1}^{n} e^{xj}}$$

Softmax transforms list of numbers into 0 and 1. Their sum is always equals to 1. We can treat those number as probabilities.

Note: log function is used optimizing DL models. Because log works better with small numbers.

Argmax and Argmin

Those two functions find the position of minimum or maximum number in a list.

Gradient Descent

Steps for how deep learning models learn:

- Guess a solution
- Compute the error
- Learn from mistakes and modify the parameters(weights and biases)

We need a mathematical description of the error and we need to find a way to minimize the error.
That is where gradient descent algorithm comes into picture.

How gradient descent algorithm works?

- Guess minimum point randomly.(doesn't matter what)
- start looping:
  Compute derivative.
  new guess = current guess - derivate scaled by learning rate.

This way we can get closer to minimum point.
Gradient descent doesn't guarantee to reach minimum point. There are some things that might go wrong:

- Poor choice of model parameters
- get stuck in a local minimum.
- vanishing gradients or exploding gradients.

Solutions for local minimum:

- Retrain model using different random weights(different starting points on the error function) and pick the model that does best.
- Increase the dimensionality(complexity) of the model to have fewer local minimum.

Gradient descent algorithm:

```
localmin = random point
learning_rate = 0.01
number_of_epochs = 100
for i in range(number_of_epochs):
(indentation)gradient = f'(localmin)
(indentation)localmin = localmin - learning_rate*gradient
```

Learning rate determines how fast we get closer to local minimum. Epoch determines number of times we run the algorithm. If learning rate is too small, epoch needs to be higher in order to reach local minimum.

Vanishing gradient: gradient becomes so small and learning doesn't continue.

Gradient: Collection of partial derivatives along different axis.

Experiment examples:
Change starting points and see final minimum point guesses.
Change learning rate and see final minimum point guesses.
Change epoch and learning rate together and see minimum point guesses.

Dynamic learning rate:
First approach, If we are far away from the minimum, gradient is large. We can take larger steps(larger learning rate) because we are far away from minimum. And if we get closer to minimum, we should get smaller steps(smaller learning rate) because we are close to minimum.
So we can update learning rate in gradient descent loop in each iteration.
learning_rate = learning_rate*np.abs(gradient) (use length of gradient if function has more than one dimension)

First approach is related to RMSProp and Adam optimizer.

Second approach,
learning_rate = learning_rate * (1-(i+1)/epochs)

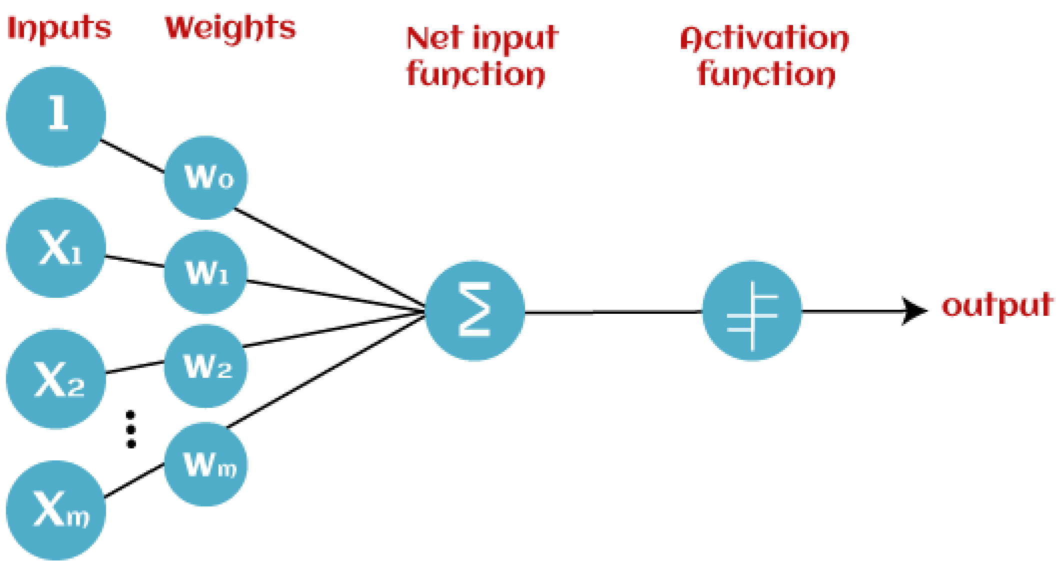
Second approach is called learning rate decay.
Summary: learning rate can change over time and it is good for optimizing the deep learning model.

Vanishing Gradient: Gradient becomes so small and we move very very slowly. We dont reach local min at the end.(Weights dont change after a point.)

Exploding Gradient: Gradient is too high. We miss the local min and end up in different point.(Weights change wildly.)

Artifical Neural Networks

Perceptron



ŷ = $\sigma(x1 * w1 + x2 * w2 + b)$

w0 is bias

Feature space: A geometric representation of the data where each feature is an axis, and each observation is a coordinate.

Loss Functions

Mean Squared Error (MSE): used for continuous data when the output is numerical prediction. ex: height, house price, temperature.

L = $\frac{1}{2}(\hat{y} - y)^2$

Cross-entropy(logistic) used for categorical data when the output is probability. ex: presence of disease, animal in picture or not, text sentiment.
L = -(ylog(ŷ) + (1-y)log(1-ŷ))

Cost Function

Cost function is average of loss functions for all data samples.

J = $\frac{1}{n}\sum_{i=1}^{n} L(\hat{y}i - yi)$

Goal of Deep Learning

Find the set of weights that minimizes the losses So we need set of weights that minimizes this function:

J = $\frac{1}{n}\sum_{i=1}^{n} L(f(x, W)i - yi)$
f corresponds to deep learning model. x is the inputs and W is the weights from the model. It produces ŷ.

Why do we train on cost and not on loss?
Training on each sample is time-consuming and may lead to overfitting.
But averaging over too many samples may decrease sensitivity.
A good solution is to train the model in batches of samples.

Back Propagation

Same as gradient descent!

$w = w - l*$ gradient of Loss or cost function
Loss function is made up by weights. We take partial derivative according to each weight and form gradient then we update weights.

Cross Validation

Before training, in order to overcome overfitting, usually data is split into three parts. %80 is for training, %10 is for devset (testing and changing parameters of the model) and %10 is for test set final test.

Note: Data should be randomized before training

How to split data before training in pytorch?

First data is split as training and test set using train_test_split() function.
Example:
train_data,test_data, train_labels,test_labels = train_test_split(data, labels, train_size=.8)
Secondly datasets are formed to combine data and labels.
train_data = torch.utils.data.TensorDataset(train_data,train_labels)
test_data = torch.utils.data.TensorDataset(test_data,test_labels)
Lastly, dataloader objects are created. They contain data and labels as batches.
train_loader = DataLoader(train_data,shuffle=True,batch_size=12)
test_loader = DataLoader(test_data,batch_size=test_data.tensors[0].shape[0])

Regularization

Prevents memorization of models. Helps the model generalize to unseen examples.
Works better for larger models with multiple hidden layers.
Generally works better with sufficient data.
There are 3 type of regularization methods.

Node Regularization: Modify the model(dropout)
Loss Regularization: Add a cost to the loss/loss function (L1/L2)
Data Regularization Modify or add data (batch training, data augmentation, normalization)

Dropout Regularization

We determine a probability(0.5 for example.) for each node to be dropped out in training.(result of that particular node of activation is forced to be zero.) Dropped out nodes change in every epoch.

In testing nodes are not dropped out.

There occurs a problem if we apply dropout method. Since we dropped out nodes during training, Input of nodes in the next layer will be decreased compared to testing(no node dropped out).

There are two approaches to solve this problem.
Multiply output of nodes(or weights?) with (1-p) during testing(we scale down to decrease input for the nodes in the next layer.)
Or we compute q = 1/1-p and multiply output of nodes(or weights?) by q during training.(we scale up to increase input for the nodes in the next layer.)
Pytorch uses second method.
Benefits of Dropout method: Prevents a single node from learning too much.Increases generalization.Works better on deep networks and lots of data.

There are two methods in pytorch. eval() and train(). Those two methods switch the model between two states. Testing(evaluation) and training. During training state dropout regularization is applied and during eval state dropout regularization is not applied. it is ignored.

L1/L2 Regularization

A small cost is added to cost function. it is used in large and complex models with lots of weights(high risk of overfitting).
L1 is used when dataset has sparse features.
L2 is used when dataset has complex features.
no need to switch between train and eval modes for this type regularization.

Note: If there are similar samples in the dataset: when batch size is smaller, model may learn faster(smaller epoch)
If batch size is larger, then model may need longer training session(more epoch) to reach a good accuracy.
If there are different samples in the dataset, reverse is true.

Metaparameters(hyperparameters)

Parameters: Features of the model that are learned by the model during training. We do not set those parameters.

Metaparameters: Features of the model that are set by humans, not learned by the model.

Some metaparameters: Model architecture, number of hidden layers,
number of units per layer, cross validation sizes, batch size, activation functions, optimization functions, learning rate, dropout, loss function, data normalization, weight normalization, weight initialization etc...

Data normalization

All samples are processed the same.
All data features are treated the same.
Weights remain numerically stable.

Z- transform formula: Zi = $\frac{xi - \hat{x}}{\sigma(x)}$
Substract mean from each individual value and divide by standart deviation.

Min-max scaling formula: $\hat{x}$ = $\frac{x - minx}{maxx - minx}$
Transforms values between 0 and 1.
If you want to scale to different range between a to b: x^* = $a + \hat{x}(b - a)$

Note: When we want to normalize data, either one is fine but generally min max scaling is common for images and uniform-data. Z-scoring is common for data that are normally distributed.

Batch Normalization

Even if the data being input to the model is normalized, data being input from one layer to another layer is not normalized. Batch normalization is normalizing data which is normalizing the output of layers.

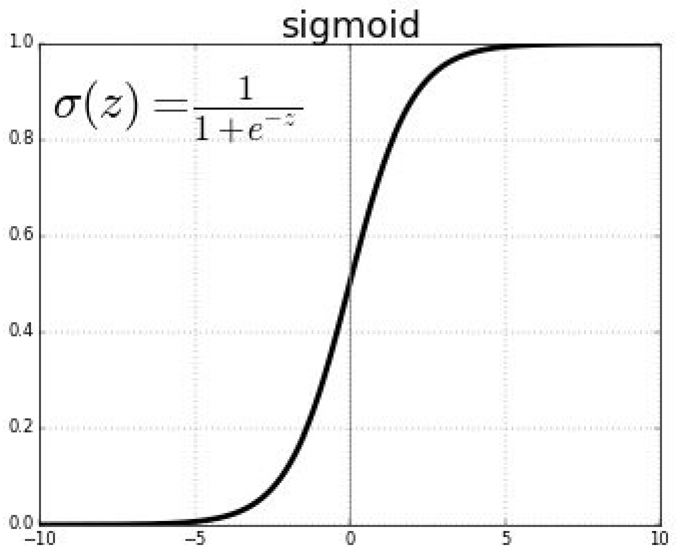
ŷ = $\sigma(W\hat{x} + b)$
$\hat{x} = \gamma * x + \beta$
γ and β are learned during training. They normalize the input to the next layer.
BatchNorm goes before the activation function.

Batch normalization should only be applied during training. It should be switched off during validation/test(model.eval())
Not: It may not be good to use dropout and batch normalization at the same time.
Batch normalization is good for networks that have lots of layers.

Activation Functions

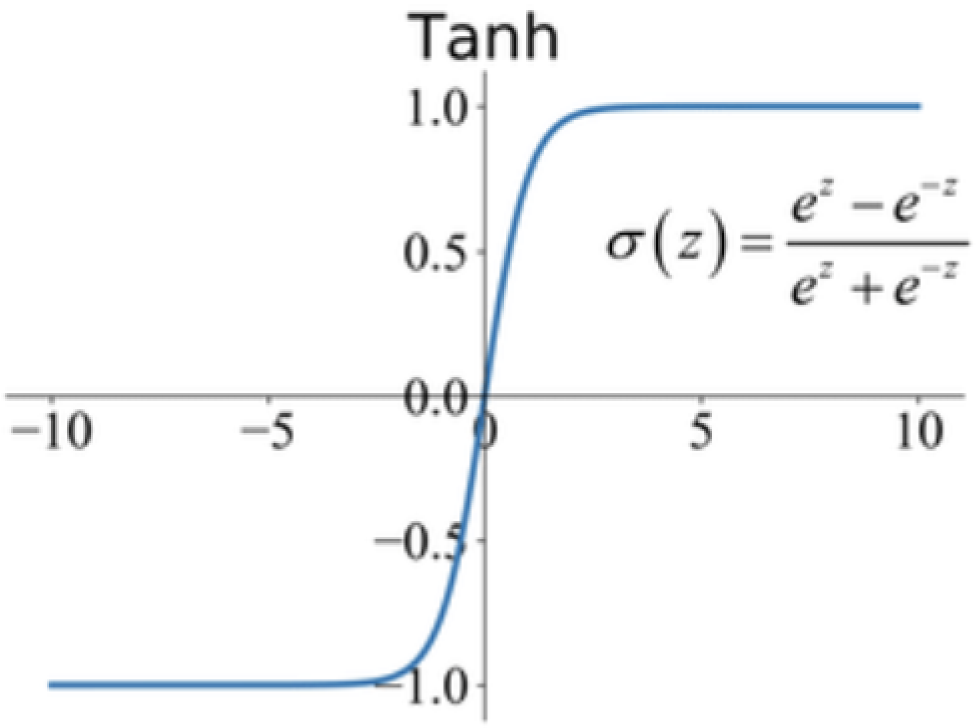
All deep learning models(no matter how deep) with linear activation functions are equal to 1 layer models.

Sigmoid Function



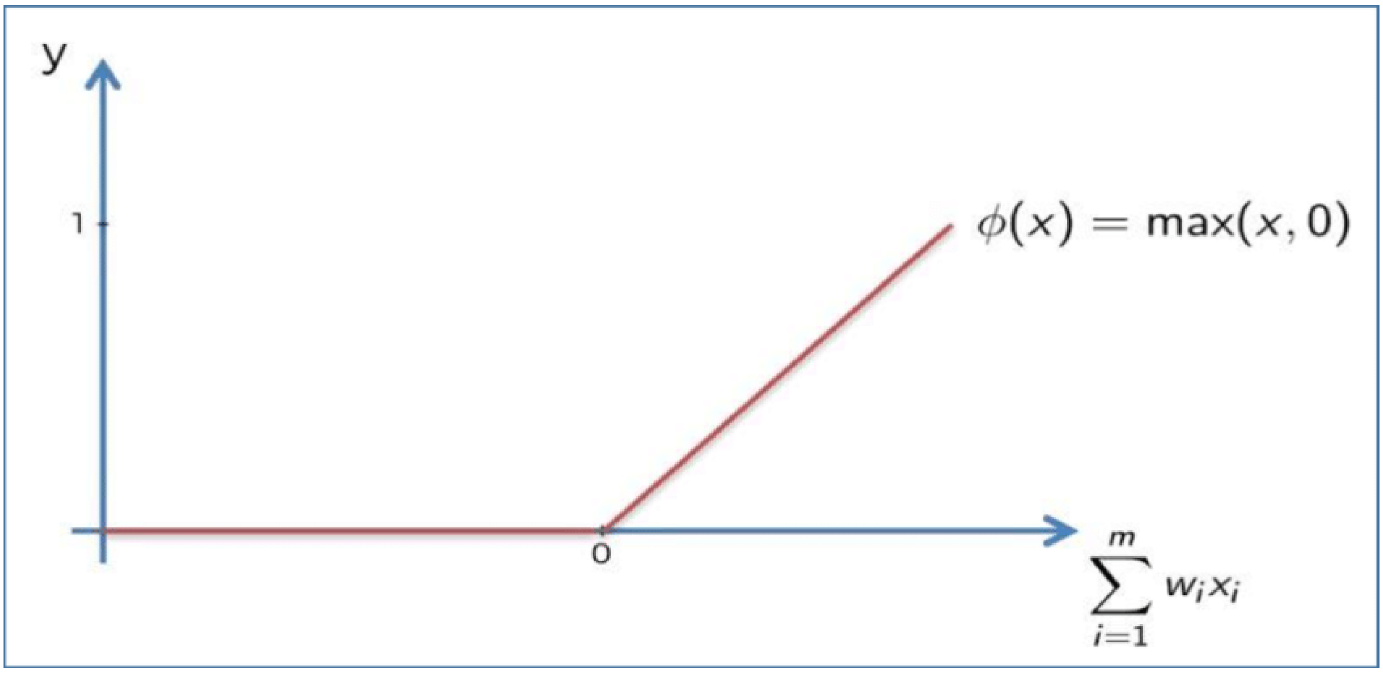
Nearly linear
Saturates between 0 and 1
Great for output.
Not preferable for hidden layers because of limited range, may lead to vanishing gradient problems
Great for binary classification.

Hyperbolic Tangent Function



Nearly linear
Saturates between -1 and 1
It is sometimes used in hidden layers but may lead to vanishing gradients.

Relu Function



Strongly nonlinear
Never saturates
Great for hidden especially if we use it with batch normalization
Easy to compute
Note: There are other version of relu function.

How to pick an activation function?
Start with commonly used functions(relu for hidden and sigmoid for output for example)
Try other functions if performence is low.

Optimizers

Algorithm that adjusts the weights during back propagation. SGD is the traditional one. It basically is the gradient algorithm. Other optimizers are modifications of SGD. Their goal is smoothing the descent.

SGD with momentum

$$vt = (1 - \beta_1)\delta L + \beta_1 Vt - 1$$
$$w = w - l * vt$$

β values are generally around .9

RMSProp

$$St = (1 - \beta_2)\delta L^2 + \beta_2 St - 1$$
$$w = w - \frac{l}{\sqrt{St + \epsilon}} * \delta L$$

Here we are changing learning rate according to the history of the magnitude of gradient.
Larger gradients = smaller learning rate(prevents exploding gradient.)
Smaller gradients = larger learning rate (prevents vanishing gradient.)

Adam (Adaptive momentum)
Combination of momentum and RMSProp. Best optimizer algorithm that exists.
$vt = (1 - \beta_1)\delta L + \beta_1 Vt - 1$
$\hat{v} = \dfrac{vt}{1 - \beta_1^t}$ (v̂ larger in the beggining and smaller at the end)
$St = (1 - \beta_2)\delta L^2 + \beta_2 St - 1$
$\check{s} = \dfrac{st}{1 - \beta_2^t}$(š larger in the beggining and smaller at the end)
$w = w - \dfrac{l}{\sqrt{\check{s} + \epsilon}} * \hat{v}$
Recommended parameters:
l = 0.001
β_1 = 0.9
β_2 = 0.999
ϵ = 10^{-8}

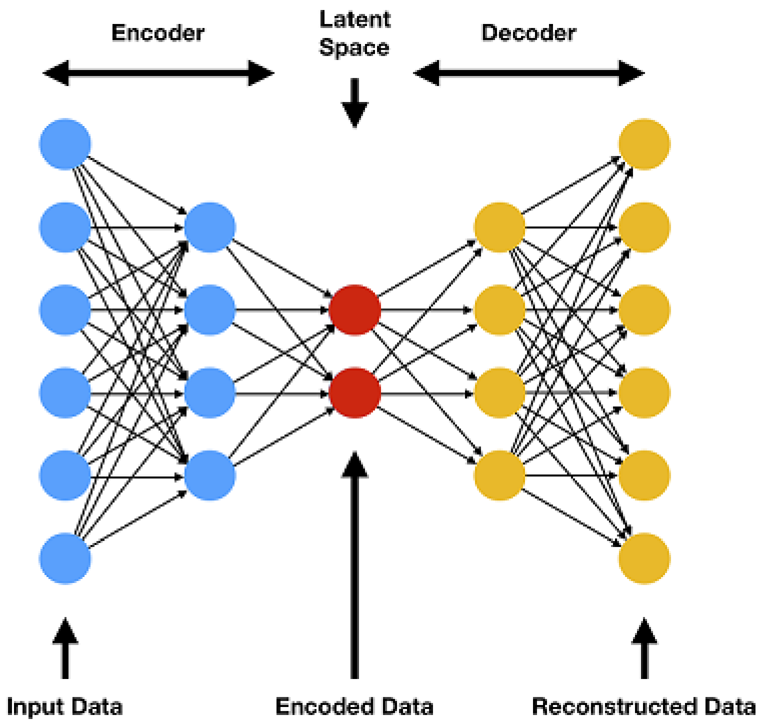
Dealing with unbalanced data

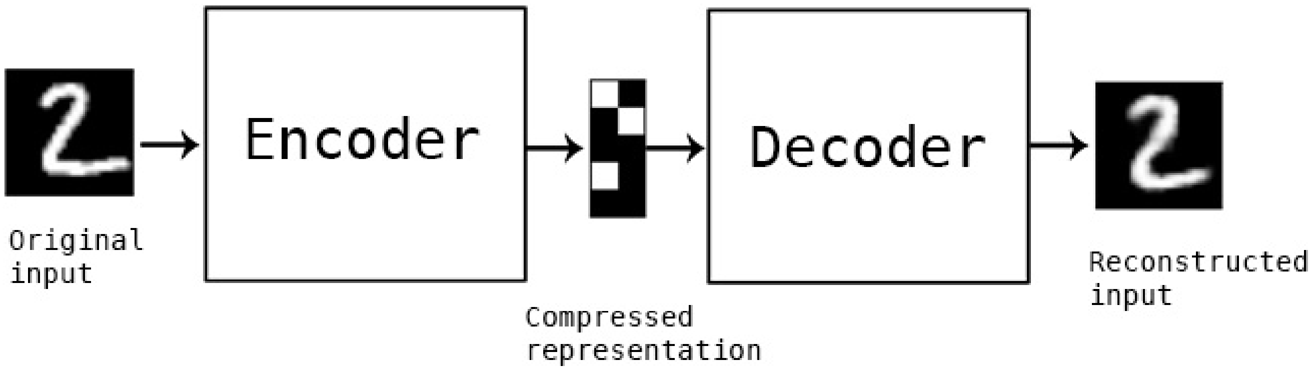
unbalanced data is when you have a large number of data samples in one category compares to others. It should be close to equal number of samples in each category.
It may occur when we want to classify rare or unusual things.
Fraud detection, disease detection etc.
Solutions:
Get more data for unbalanced category.
Undersample: Throw data in the category that has more data to match sample sizes. It works good when you have lots of data.
Oversample:Create multiple copies of the data(increases overfitting.)
Note: traning data should be oversampled after splitting data. If data is oversampled before splitting, there is a risk of copies of the same data might be both in the test and training set.
Data Augmentation: Create new data by adding new features or changing features. Used usually for images

Dealing with missing data in the dataset

Remove the row completely from the dataset
Replace missing data with the mean value of the column.
Predict missing value using machine learning or deep learning techniques.

Autoencoders





Autoencoders are used for:

Data compression or dimension reduction.

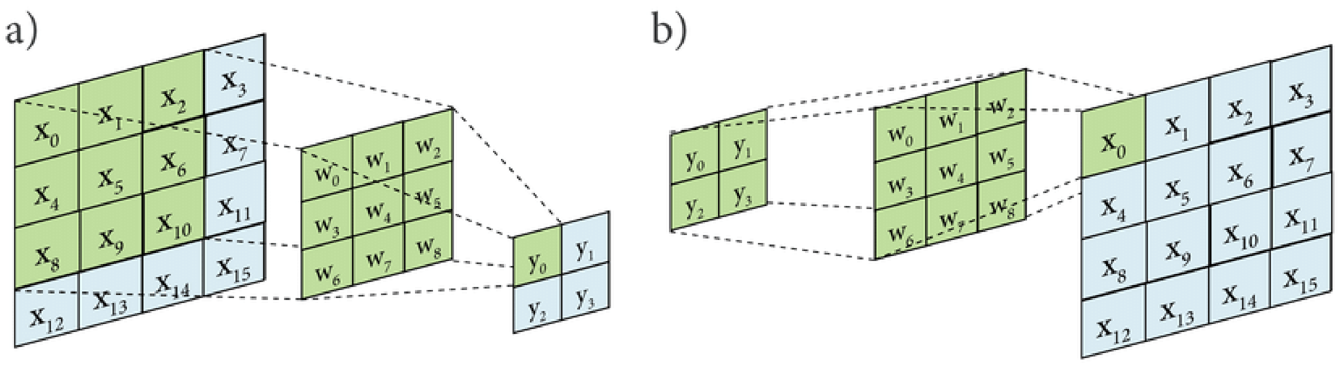
Data cleanining(denoising,despeckling,occlusion)

Feature extraction

Anomaly/fraud detection

Pretraining deep or complex models

Convolutional Neural Networks(CNN)



Kernels are filters that extract features from an image. Kernels are generally small (3x3,5x5,7x7 etc)

In deep learning, kernels are random and being learned through gradient descent algorithm. After learning same kernel is used for all images.
Kernels are not used to classify or make decisions. They are used to extract features. Those features are used for classification.

Padding: used to increase size of the result of convolution. Basically we add zeros around the image.
Stride: used to decrease the size of the result of convolution. It's a mechanism of downsampling. Reduces the parameters in CNN.The stride parameter is integer. 1,2 for instance. 1 is the default. This parameter determines how much the kernel moves along vertically and horizontally when convolution is being calculated. Different stride can be set for column and row move.

Formula for number of pixels in current layer(for height)
N_h = $\lfloor \frac{M_h + 2p - k}{S_h} \rfloor + 1$

N_h= number of pixels in the current layer
M_h = number of pixels in the previous layer.
p = padding
k = kernel
S_h = stride

Transpose Convolution

Means Scalar multiply a kernel by each pixel in an image. used for autoencoders and super resolution CNN. Its upsampling. Result will be higher resolution than original image.

Average/Mean Pooling

Values of kernel is equal to each other and their sum is one. Kernel moves on the nonoverlapping part of the image(Like stride is 2 if kernel is 2x2.) and convolution is calculated. Seems that it is same as calculating mean of part of the image where kernel overlaps.

Highlights sharp features. Useful for sparse data and increasing contrast.

Max Pooling

There is no kernel values for this operation. Mechanism works same way as average pooling. Kernel selects max value
Smooths images. Useful for noisy data and reduces impact of outliners on learning.
There is also min pooling. Works same way as max pooling but selects min.

Why use pooling?
Pooling reduces dimensionality(fewer parameters)

In CNN, deeper into the model we want more channels with fewer pixels
Selects for features over a broader spatial area(increased receptive field size)

Anns have one to one mapping between pixel and unit. In other words, the receptive field of an ANN unit is one pixel.This makes the model unable to recognize translations, resizing, rotations etc. We want a model that can see the features anywhere in the image.
With the help of pooling, we reduce the dimensionality of the image in each layer, and each layer sees more of the image as you go deep into the network.

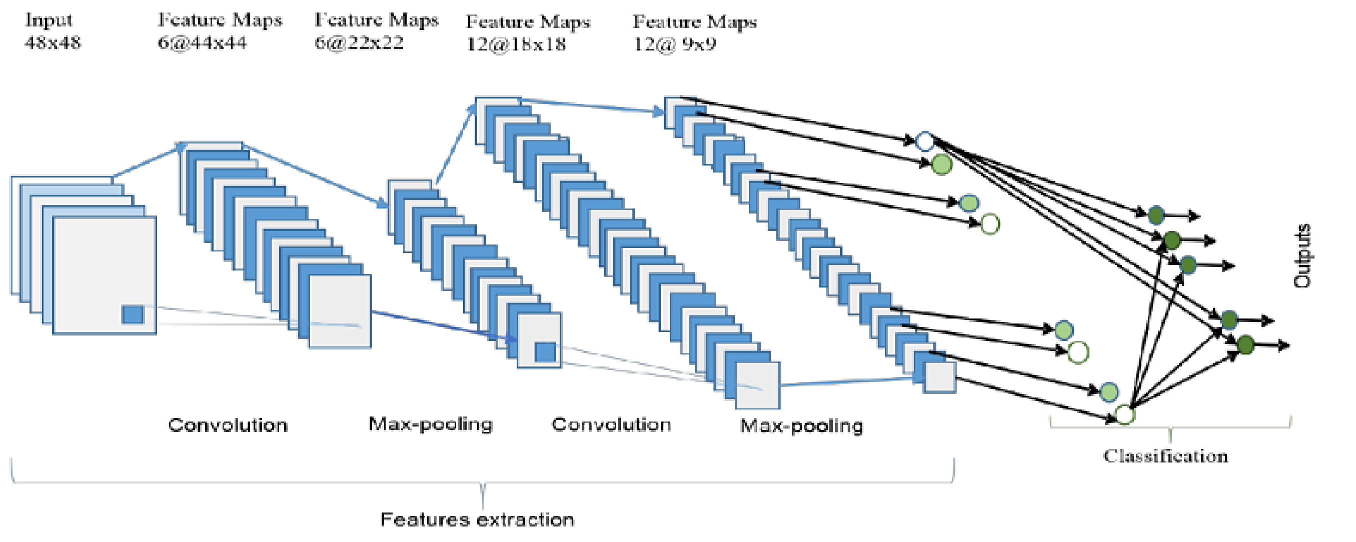


Fig. 11 The overall architecture of the CNN includes an input layer, multiple alternating convolution and max-pooling layers, one fully-connect

Image transforms

Two reasons to transform images: Pre trained CNNs are trained for certain image sizes. It may be needed to resize the images or to convert to grayscale.
Transforming images changes raw pixel values without changing the image information. So it may be useful to increase total amount of data(data augmentation.)

We need to create our own DataSet class for image transforms. Because DataSet class that comes from torch library doesn't support transforms.
Transformations are defiend in our own custom DataSet class which inherits from original DataSet class and transformations are applied when we pull data from DataLoader object(**getitem** function is being called.)

Steps:
Import the data
Create Custom DataSet Class
Define transformations
Create a DataSet with your data and transformation
Create a DataLoader(same as usual)

class customDataset(Dataset):
def **init**(self, tensors, transform=None):

```
# check that sizes of data and labels match
assert all(tensors[0].size(0)==t.size(0) for t in tensors), "Size mismatch
between tensors"

# assign inputs
self.tensors   = tensors
self.transform = transform
```

def **getitem**(self, index):

```
# return transformed version of x if there are transforms
if self.transform:
  x = self.transform(self.tensors[0][index])
else:
  x = self.tensors[0][index]

# and return label
y = self.tensors[1][index]

return x,y # return the (data,label) tuple
```

def **len**(self):
return self.tensors[0].size(0)

imgtrans = T.Compose([
T.ToPILImage(),
T.RandomVerticalFlip(p=.5),
# T.RandomRotation(90),
T.ToTensor()
])
(Some transformations require images to be in PILIImage type so image first converted to PILIImage and then transformations are applied. And lastly, ToTensor converts images into tensor and also normalize.)

train_data = customDataset((dataT,labelsT),imgtrans)
dataLoaded = DataLoader(train_data,batch_size=8,shuffle=False)

Architecture of CNNs

There are three types of layers in the CNN.
Convolutional layer: Kernels are learned through backpropagation to create feature maps.
Pooling: Reduce dimensionality and increase receptive field size.
Fully connected: Traditional ANN. Categorical or continuous prediction.

As we go deeper into the network, image resolution(number of pixels in each layer) decreases and number of filters increase(receptive field.)

Note: Usually, dropout is applied to convolutional layers with propability like 0.1 or 0.2. And for feed forward part you can go up to 0.5 or a little bir higher.