UNIVERSITY OF SCIENCE
Ho Chi Minh City

INFORMATION OF TECHNOLOGY
High quality

A
Project
Report
On

NEURAL NETWORK

Under the course of

CSC14003 – Artificial Intelligence

Semester II

Sumitted by:
Class:19CLC9

1.	Ho Ngoc Minh Duc	19127368
2.	Nguyen Phuong Vy	19127088
3.	Nguyen Kim Thi To Nga	19127219

Dr. Chau Thanh Duc

Nguyen Van Quang Huy **Phan Thi Phuong Uyen**

Academic year
(2019-2023)

Contents

Working Detail Table

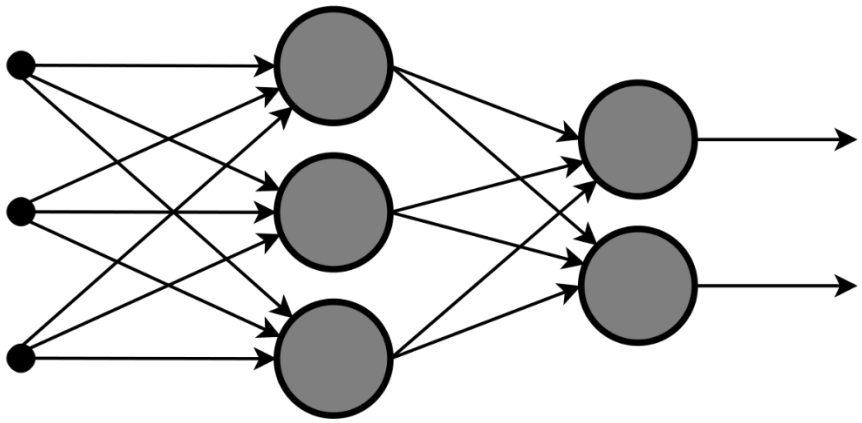
Ho Ngoc Minh Duc	19127368	Design model, design app
Nguyen Phuong Vy	19127088	Analysis model, app for part II and III, summarize information.
Nguyen Kim Thi To Nga	19127219	Find information for Theory part

I. Theory

1. What is an Artificial Neural Network?

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



An example of neural network.

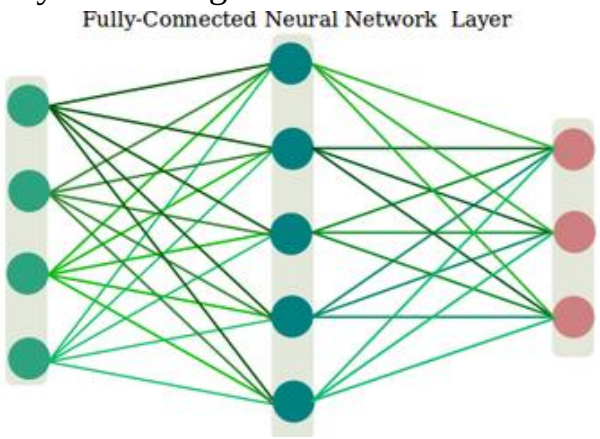
2. Neural Network Layer Types

Different layers perform different transformations on their inputs, and some layers are better suited for some tasks than others.

- ***Fully connected layer***

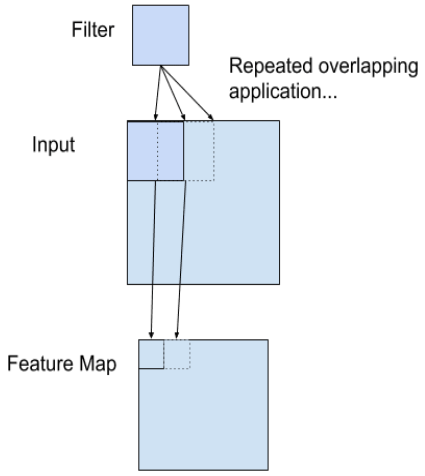
  In a neural network model, every hidden layer is called a fully connected layer. The whole model is called a fully connected neural network (FCN)
  A fully connected neural network consists of a series of fully connected layers.
  A fully connected layer is a function from $\mathbb{R}_m$to $\mathbb{R}_n$. Each output dimension depends on each input dimension.

It is used for giving answers. For example, the output from the convolutional layers represents high-level features in the data. While that output could be flattened and connected to the output layer, adding a fully connected layer is a (usually) cheap way of learning non-linear combinations of these features.



A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.

❖ *Convolutional layers*



Example of a Filter Applied to a Two-Dimensional Input to Create a Feature Map.

Convolutional layers are the major building blocks used in convolutional neural networks.

A convolution is the simple application of a filter to an input that results in an activation. Repeated application of the same filter to an input results in a map of activations called a feature map, indicating the locations and strength of a detected feature in an input, such as an image.

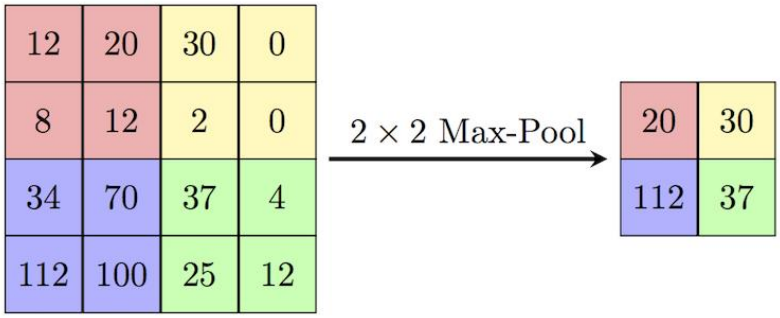
The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel specific to a training dataset under the constraints of a specific predictive modeling problem, such as image classification. The result is highly specific features that can be detected anywhere on input images.

❖ *Pooling layers*

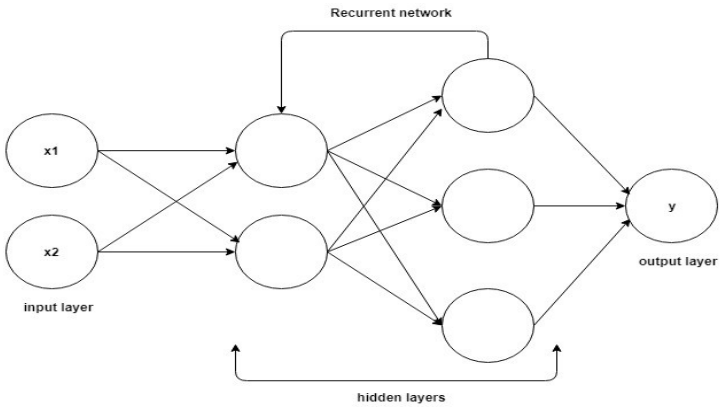
Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.

The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

The most common approach used in pooling is max pooling.



❖ *Recurrent layers*



In recurrent layer (also call RNN-Recurrent Neural Network), output from previous step are fed as input to the current step.

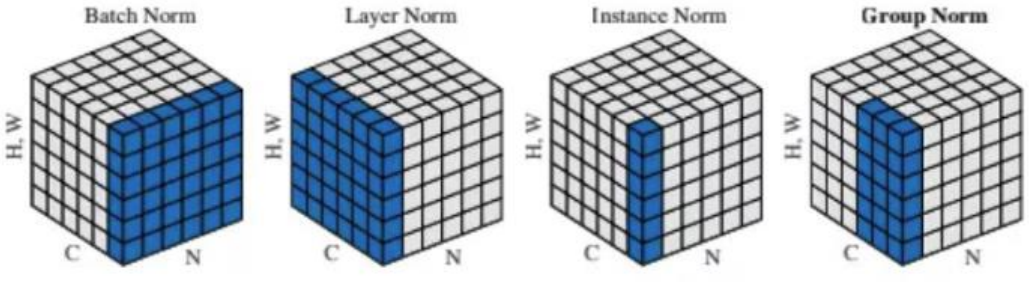
In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

Recurrent layer have a "memory" which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

❖ *Normalization layers*

Training state-of-the-art, deep neural networks is computationally expensive. One way to reduce the training time is to normalize the activities of the neurons. A recently introduced technique called batch normalization uses the distribution of the summed input to a neuron over a mini-batch of training cases to compute a mean and variance which are then used to normalize the summed input to that neuron on each training case. This significantly reduces the training time in feedforward neural networks. However, the effect of batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to recurrent neural networks.

Layer normalization is a simple normalization method to improve the training speed for various neural network models. Unlike batch normalization, the proposed method directly estimates the normalization statistics from the summed inputs to the neurons within a hidden layer so the normalization does not introduce any new dependencies between training cases. Layer normalization works well for RNNs and improves both the training time and the generalization performance of several existing RNN models.



A visual comparison of various normalization methods

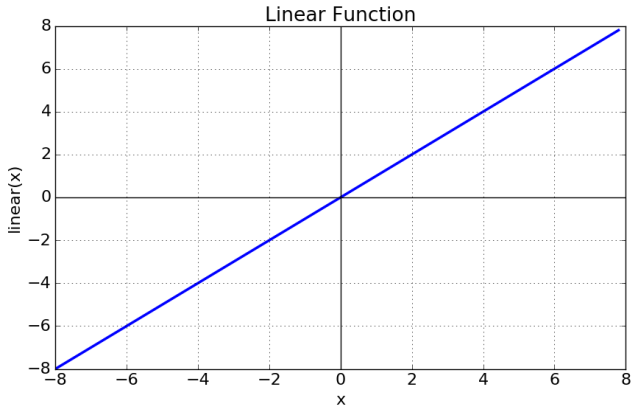
3. Activation Functions

Activation Function is just function that you use to get the output of node, which is also known as **Transfer Function**.

The Activation Functions can be basically divided into 2 types:

❖ *Linear Activation Function*

As you can see the function is a line or linear. Therefore, the output of the functions will not be confined between any range.



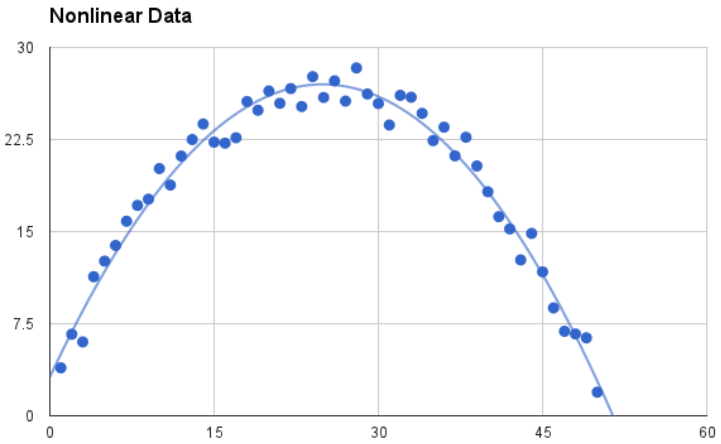
Equation: f(x) = x

Range: (-infinity to infinity)

It doesn't help with the complexity or various parameters of usual data that is fed to the neural networks.

❖ *Non-linear Activation Functions*

The Nonlinear Activation Functions are the most used activation functions. Nonlinearity helps to makes the graph look something like this:



It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.

The main terminologies needed to understand for nonlinear functions are:
Derivative or Differential: *Change in y-axis w.r.t. change in x-axis. It is also known as slope.*
Monotonic function: *A function which is either entirely non-increasing or non-decreasing.*

The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves:**

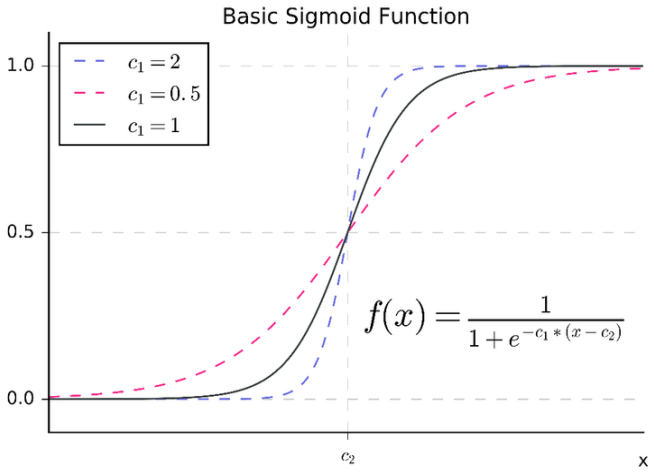
- **Sigmoid Activation Function:**

The Sigmoid Function have a characteristic S-shape curve or sigmoid curve.

Range: (0;1)

The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

A sigmoid function is a **bounded**, **differentiable**, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point. A sigmoid "function" and a sigmoid "curve" refer to the same object.



- **Softmax Activation Function:**

The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems.

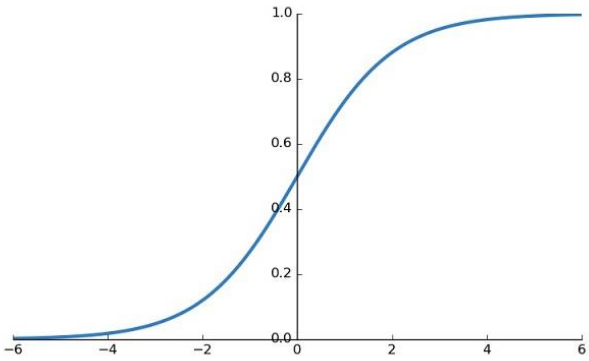
Range: (0;1]

Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

The softmax function turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as "probabilities". If one of the input is small or negative, the softmax transforms it into a small probabilities, and if an input is large, then it turns it into a large probability, but it will always remain in range (0,1].

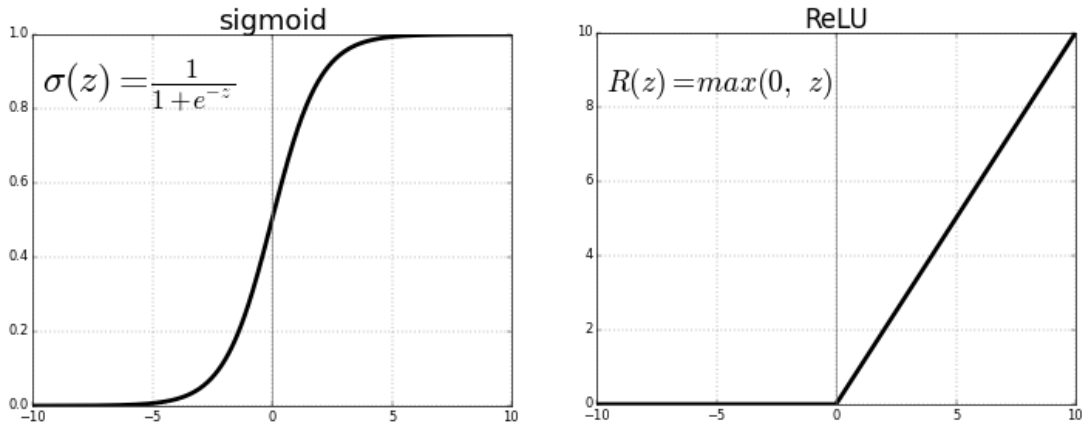
Formula of Softmax Function:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$



- **ReLU (Rectified Linear Unit) Activation Function:**

The ReLU is the most used activation function in the world right now. Since it is used in almost all the convolutional neural networks or deep learning.



ReLU vs. Logistic Sigmoid

As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

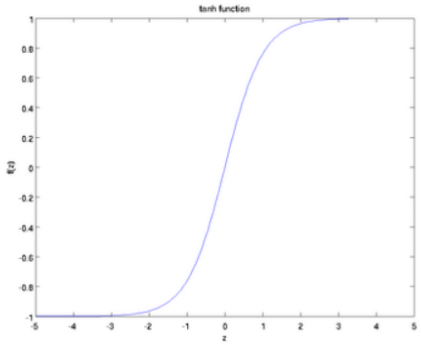
Range: [0 to infinity)

The function and its derivative both are monotonic.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately.

- **Tanh Function**

Another activation function that is used is the tanh function.



$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!

$$tanh(x) = 2\,sigmoid(2x) - 1$$

This has characteristics similar to sigmoid that we discussed above. It is nonlinear in nature, so great we can stack layers! It is bound to range (-1, 1) so no worries of activations blowing up. One point to mention is that the gradient is stronger for tanh than sigmoid (derivatives are steeper). Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Like sigmoid, tanh also has the vanishing gradient problem.

Tanh is also a very popular and widely used activation function.

4. Loss Functions

The loss function is the bread and butter of modern machine learning; it takes your algorithm from theoretical to practical and transforms neural networks from glorified matrix multiplication into deep learning.

Neural Network uses optimising strategies like stochastic gradient descent to minimize the error in the algorithm. The way we actually compute this error is by using a Loss Function.

At its core, a loss function is incredibly simple: It's a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you're getting anywhere.

❖ *Cross-entropy Loss Function*

When working with Neural Network, the objective is almost always to minimize the loss function. The lower the loss the better the model. Cross-entropy loss is a most important loss function. It is used to optimize classification models. The understanding of Cross-entropy is pegged on understanding of Softmax activation function.

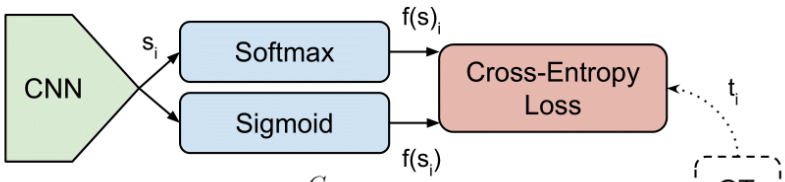
The Cross-Entropy Loss is defined as: $CE = -\sum_i^C t_i log\,(s_i)$

Where t_i and s_i the groundtruth and the CNN score for each class i in C. As usually an activation function (Sigmoid / Softmax) is applied to the scores before the CE Loss computation, we write $f(s_i)$ to refer to the activations.

In a binary classification problem, when $C' = 2$, the Cross-Entropy Loss can be defined also as:

$$CE = -\sum_{i=1}^{C'=2} t_i \log(s_i) = -t_1 \log(s_1) - (1 - t_1)\log(1 - s_1)$$

Figure of Cross-Entropy Loss:



$$CE = -\sum_{i}^{C} t_i log(f(s)_i)$$

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

Logistic Loss and Multinomial Logistic Loss are other names for Cross-Entropy loss.

❖ Mean Squared Error

Mean Squared Error (MSE) is the mean of squared differences between the actual and predicted value. If the difference is large the model will penalize it as we are computing the squared difference.

MSE is a risk function, corresponding to the expected value of the squared error loss.

The MSE is a measure of the quality of an estimator - it is always non-negative, and values closer to zero are better (but not zero).

The MSE either assesses the quality of a predictor or of an estimator. The definition of an MSE differs according to whether one is describing a predictor or an estimator.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

5. Metric algorithms

A metric is a function that is used to judge the performance of your model.

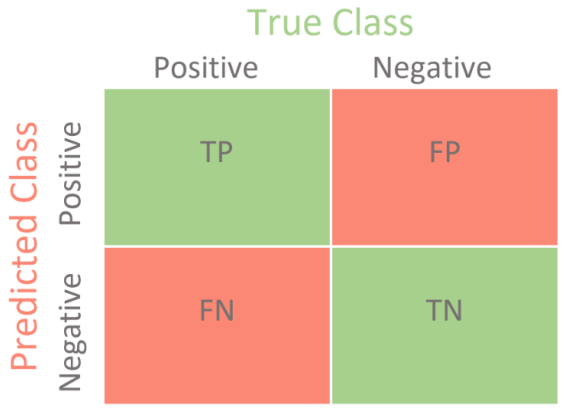
Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Note that you may use any loss function as a metric.

❖ *Confusion Matrix*

A holistic way of viewing true and false postitive and negative results is with a confusion matrix. Despite the name, it is a straightforward NxN matrix that provides an intuitive summary of the inputs to the calculations that we made above. Rather than a decimal correctness, the confusion matrix gives us counts of each of the types of results.

The table above describes an output of negative vs. positive. These two outcomes are the "classes" of each examples. Because there are only two classes, the model used to generate the confusion matrix can be described as a binary classifier. (Example of a binary classifier: spam detection. All emails are spam or not spam.)

For a binary classification problem, we would have a 2 x 2 matrix as shown below with 4 values:



Let's decipher the matrix:

- The target variable has two values: **Positive** or **Negative**
- The **columns** represent the **actual values** of the target variable
- The **rows** represent the **predicted values** of the target variable

Understanding True Positive, True Negative, False Positive and False Negative in a Confusion Matrix:

True Positive (TP)

- The predicted value matches the actual value
- The actual value was positive and the model predicted a positive value

True Negative (TN)

- The predicted value matches the actual value
- The actual value was negative and the model predicted a negative value

False Positive (FP) – Type 1 error

- The predicted value was falsely predicted
- The actual value was negative but the model predicted a positive value
- Also known as the **Type 1 error**

False Negative (FN) – Type 2 error

- The predicted value was falsely predicted
- The actual value was positive but the model predicted a negative value
- Also known as the **Type 2 error**

❖ *Accuracy*

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

Accuracy is the ratio of number of correct predictions to the total number of input samples. It is an incredibly straightforward measurement, and thanks to its simplicity it is broadly useful.

For example, consider that there are 98% samples of class A and 2% samples of class B in our training set. Then our model can easily get 98% training accuracy by simply predicting every training sample belonging to class A.

When the same model is tested on a test set with 60% samples of class A and 40% samples of class B, then the test accuracy would drop down to 60%. Accuracy is great, but gives us the false sense of achieving high accuracy.

❖ *Precision*

When the model predicts positive, how often is it correct?

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Precision is a similar metric, but it only measures the rate of false positives, it helps when the costs of false positives are high. In certain domains, like spam detection, a false positive is a worse error than a false negative (generally, missing an important email is worse than the inconvenience of deleting a piece of spam that snuck through the filter).

When Precision = 1, this means that all samples are positive and none of them are classified incorrectly. But Precision = 1 doesn't ensure whether the model has found all the positive samples. If a model only found 1 positive sample that it is certain, we can't call that a good model.

❖ *Recall*

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Recall is the opposite of precision, it measures false negatives against true positives. False negatives are especially important to prevent in disease detection and other predictions involving safety. Recall helps when the cost of false negatives is high.

When Recall = 1, this means every positive samples are found. However, this value doesn't measure how many negative in the result. If the model classify every samples are positive, the Recall definitely equals 1, yet we still see this is the worst model.

=> A good-classified model is the one that has both high Presicion and Recall, which the closer to 1 the better

❖ *F1 Score*

What if you want to balance the two objectives: high precision and high recall?

We calculate the F1-score as the harmonic mean of precision and recall to accomplish just that. While we could take the simple average of the two scores, harmonic means are more resistant to outliers. Thus, the F1-score is a balanced metric that appropriately quantifies the correctness of models across many domains.

$$F1 = 2 \times \frac{Precision\ \times Recall}{Precision + Recall}$$

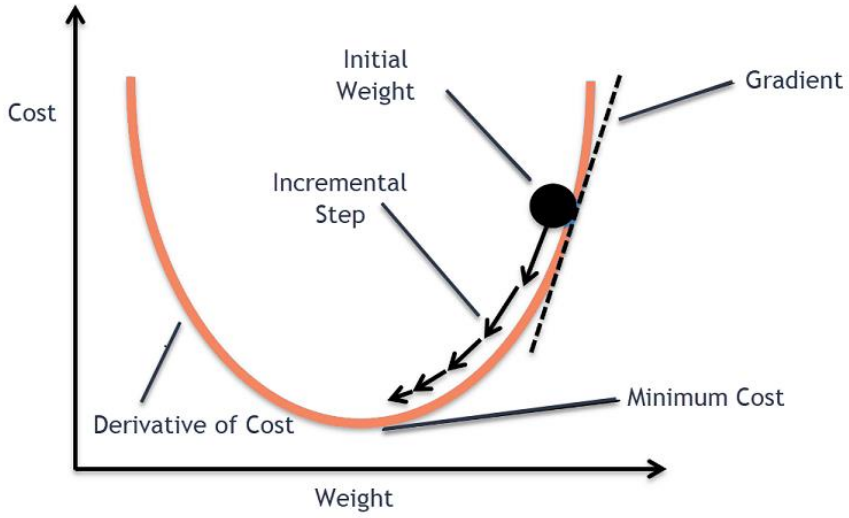
F1 is an overall measure of a model's accuracy that combines precision and recall, in that weird way that addition and multiplication just mix two ingredients to make

a separate dish altogether. That is, a good F1 score means that you have low false positives and low false negatives, so you're correctly identifying real threats and you are not disturbed by false alarms. An F1 score is considered perfect when it's 1, while the model is a total failure when it's 0.

6. Gradient descent

Gradient descent is an optimization algorithm that's used when training a machine learning model. It's based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum.

A gradient simply measures the change in all weights with regard to the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.



Gradient descent

- **How gradient descent work?**

Consider gradient descent as hiking down to the bottom of a valley rather than climbing up a hill. This is a better analogy because it is a minimization algorithm that minimizes a given function.

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

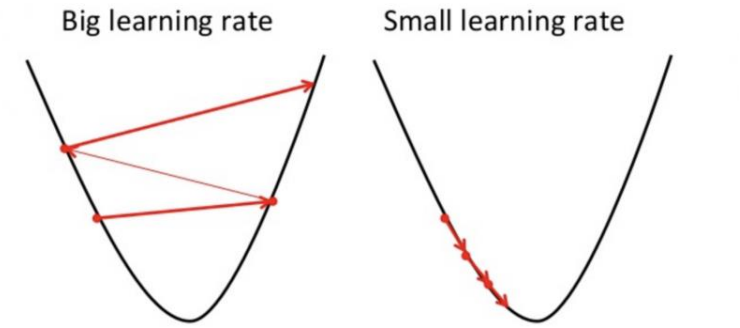
This equation describes how gradient descent does: a_{n+1} is the next position of our climber, while a_n represents his current position. The minus sign refers to the

minimization part of gradient descent. The gamma in the middle is a waiting factor and the gradient term ($\Delta f(a)$) is simply the direction of the steepest descent.

So this formula basically tells us next position we need to go, which is the direction of the steepest descent.

- **Learning rate** (η)

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image). So the learning rate should never be too high or too low for this reason.



- **Cost function**

A good way to make sure gradient descent runs properly is by plotting the cost function as the optimization runs. Put the number of iterations on the x-axis and the value of the cost-function on the y-axis. This helps you see the value of your cost function after each iteration of gradient descent and provides a way to easily spot how appropriate your learning rate is.

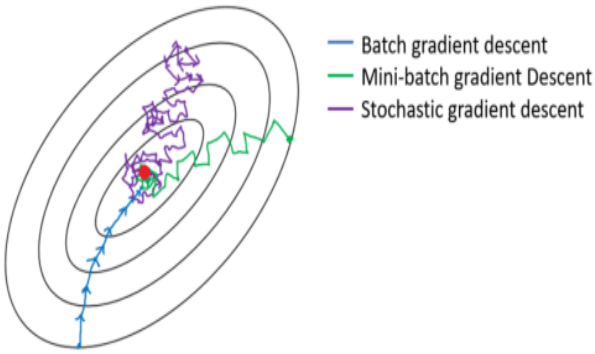
A Loss Functions tells us "how good" our model is at making predictions for a given set of parameters. The cost function has its own curve and its own gradients. The slope of this curve tells us how to update our parameters to make the model more accurate.

If gradient descent is working properly, the cost function should decrease after every iteration.

• **Type of Gradiant Descent:** there are some popular types of gradient descent that mainly differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update:

A. Batch gradient descent:

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.



In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters θ. So that's just one step of gradient descent in one epoch.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta).$$

Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.

Advantages of Batch Gradient Descent

- Less oscillations and noisy steps taken towards the global minima of the loss function due to updating the parameters by computing the average of all the training samples rather than the value of a single sample.
- It can benefit from the vectorization which increases the speed of processing all training samples together.
- It produces a more stable gradient descent convergence and stable error gradient than stochastic gradient descent.
- It is computationally efficient as all computer resources are not being used to process a single sample rather are being used for all training samples.

Disadvantages of Batch Gradient Descent

- Sometimes a stable error gradient can lead to a local minima and unlike stochastic gradient descent no noisy steps are there to help get out of the local minima.
- The entire training set can be too large to process in the memory due to which additional memory might be needed.
- Depending on computer resources it can take too long for processing all the training samples as a batch.

B. Stochastic gradient descent:

Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$. Depending on the problem, this can make SGD faster than batch gradient descent. One advantage is the frequent updates allow us to have a pretty detailed rate of improvement.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Advantages of Stochastic Gradient Descent

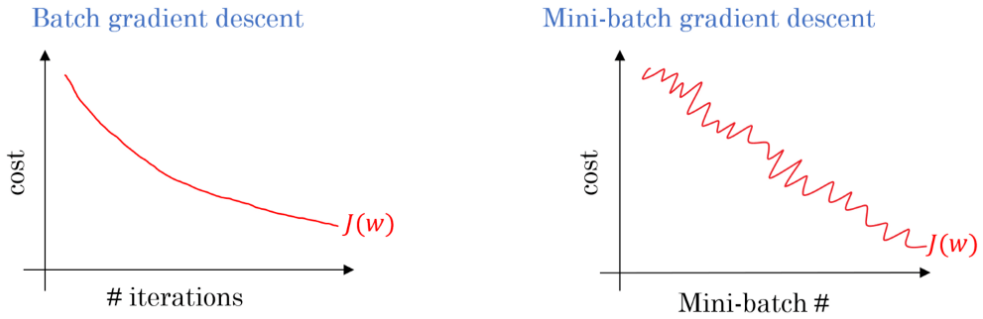
- It is easier to fit into memory due to a single training sample being processed by the network.
- It is computationally fast as only one sample is processed at a time.
- For larger datasets it can converge faster as it causes updates to the parameters more frequently.
- Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums of the loss function (in case the computed position turns out to be the local minimum)

Disadvantages of Stochastic Gradient Descent

- Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions.
- Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function
- Frequent updates are computationally expensive due to using all resources for processing one training sample at a time.
- It loses the advantage of vectorized operations as it deals with only a single example at a time.

C. Mini Batch gradient descent:

This is a mixture of both stochastic and batch gradient descent. The training set is divided into multiple groups called batches. Each batch has a number of training samples in it. At a time a single batch is passed through the network which computes the loss of every sample in the batch and uses their average to update the parameters of the neural network.



$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}).$$

Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini batches of fixed size, we do the following steps in **one epoch:**

1. Pick a mini batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

This ensures the following advantages of both stochastic and batch gradient descent are used due to which Mini Batch Gradient Descent is most commonly used in practice.

- Easily fits in the memory.
- It is computationally efficient.
- Benefit from vectorization.

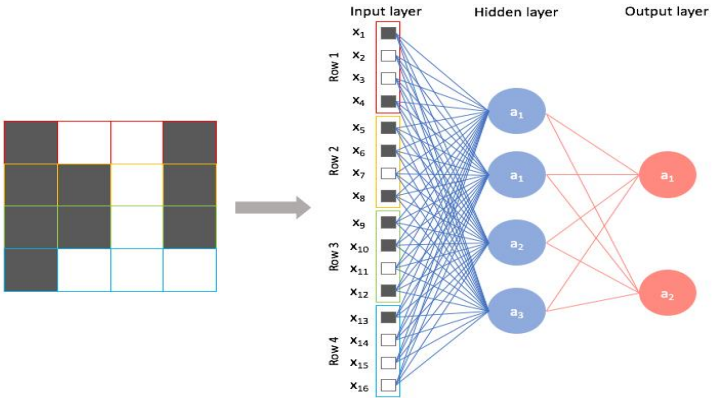
- If stuck in local minimums, some noisy steps can lead the way out of them.
- Average of the training samples produces stable error gradients and convergence.

7. Feed Forward and Back Propagation

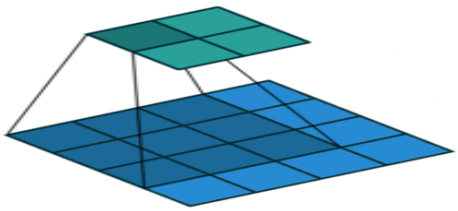
➢ Convolutional layers

❖ *Feed forward*

First, let's examine what this would look like using a feed-forward network.

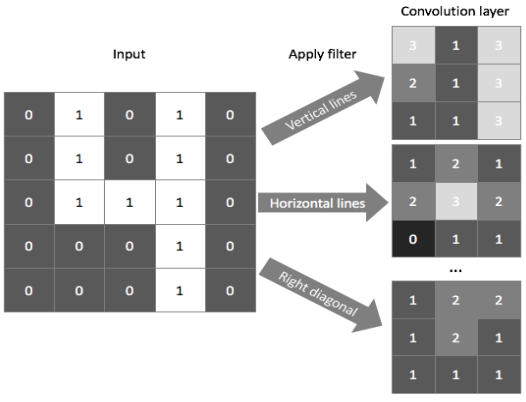


A feed-forward network takes a vector of inputs, so we must flatten our 2D array of pixel values into a vector
Now, we can treat each individual pixel value as a feature and feed it into the neural network.

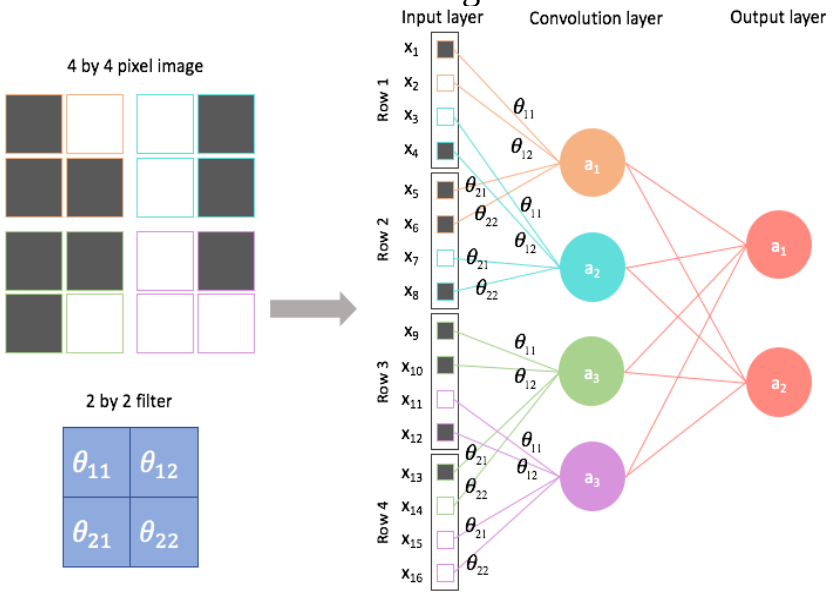


In Convolution layers, we'll scan each filter across the image calculating the linear combination (and subsequent activation) at each step. You can imagine each pixel in the convolved layer as a neuron which takes all of the pixel values *currently in the window* as inputs, linearly combined with the corresponding weights in our filter. We overlay this filter onto the image and linearly combine (and then activate) the pixel and filter values.

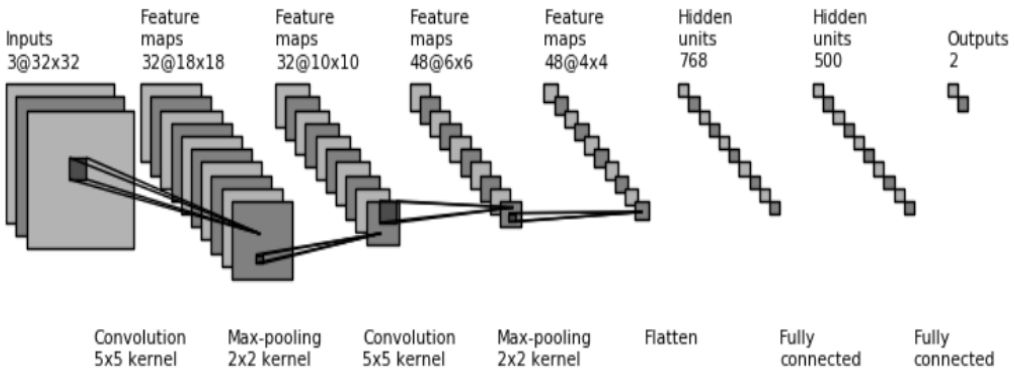
We can stack layers of convolutions together (ie. perform convolutions on convolutions) to learn more intricate patterns within the features mapped in the previous layer. This allows our neural network to identify general patterns in early layers, then focus on the patterns within patterns in later layers.



In this case, a 2 by 2 filter with a stride of 2 (more on strides below) is scanned across the image to output 4 nodes, each containing localized information about the image.

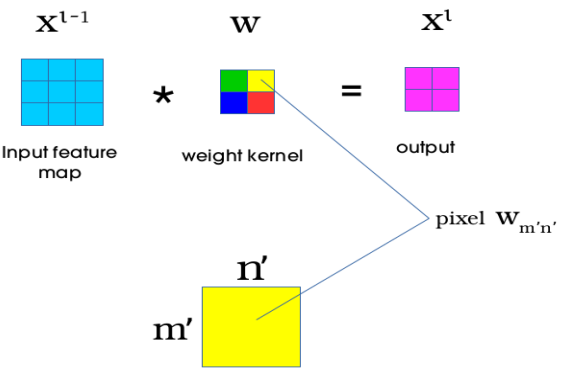


❖ ***Backward***



For backpropagation there are two updates performed, for the weights and the deltas. Lets begin with the weight update.

We are looking to compute $\frac{\partial E}{\partial w^{l}_{m',n'}}$ which can be interpreted as the measurement of how $w_{m',n'}$ change in a single pixel in the weight kernel affects the loss function E.

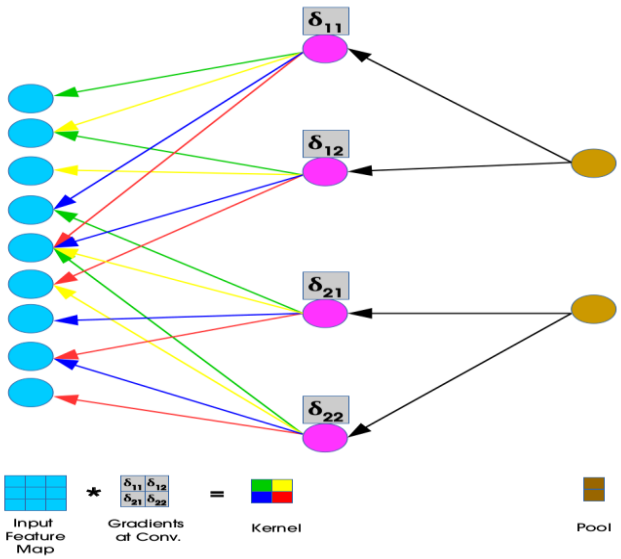


During forward propagation, the convolution operation ensures that the yellow pixel in the weight kernel makes a contribution in all the products (between each element of the weight kernel and the input feature map element it overlaps). This means that pixel will eventually affect all the elements in the output feature map.

The gradient component for the individual weights can be obtained by applying the chain rule in the following way:

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$
$$= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$

The diagram below shows gradients (δ11,δ12,δ21,δ22) generated during backpropagation:



During the reconstruction process, the deltas (δ11,δ12,δ21,δ22) are used. At this point we are looking to compute $\frac{\partial E}{\partial x^l_{i',j'}}$ which can be interpreted as the measurement of how the change in a single pixel xi′,j′ in the input feature map affects the loss function E.

Using chain rule and introducing sums give us the following equation:

$$\frac{\partial E}{\partial x^l_{i',j'}} = \sum_{i,j \in Q} \frac{\partial E}{\partial x^{l+1}_Q} \frac{\partial x^{l+1}_Q}{\partial x^l_{i',j'}}$$
$$= \sum_{i,j \in Q} \delta^{l+1}_Q \frac{\partial x^{l+1}_Q}{\partial x^l_{i',j'}}$$

➢ Pooling Layer

The function of the pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. No learning takes place on the pooling layers.

Pooling units are obtained using functions like max-pooling, average pooling and even L2-norm pooling. At the pooling layer:

- **Forward propagation** results in an N×N pooling block being reduced to a single value - value of the "winning unit".
- **Backpropagation** of the pooling layer then computes the error which is acquired by this single value "winning unit". To keep track of the "winning unit" its index noted during the forward pass and used for gradient routing during backpropagation. Gradient routing is done in the following ways:
  - Max-pooling - the error is just assigned to where it comes from - the "winning unit" because other units in the previous layer's pooling blocks did not contribute to it hence all the other assigned values of zero
  - Average pooling - the error is multiplied by $\frac{1}{N \times N}$ and assigned to the whole pooling block (all units get this same value).

➢ Conclusion

Convolutional neural networks employ a weight sharing strategy that leads to a significant reduction in the number of parameters that have to be learned. The presence of larger receptive field sizes of neurons in successive convolutional layers coupled with the presence of pooling layers also lead to translation invariance. As we have observed the derivations of forward and backward propagations will differ depending on what layer we are propagating through.

8. Optimizers

Optimizers are algorithm or methods used to change the attributes of your nearal network such as weights and learnings rate in order to reduce the losses.

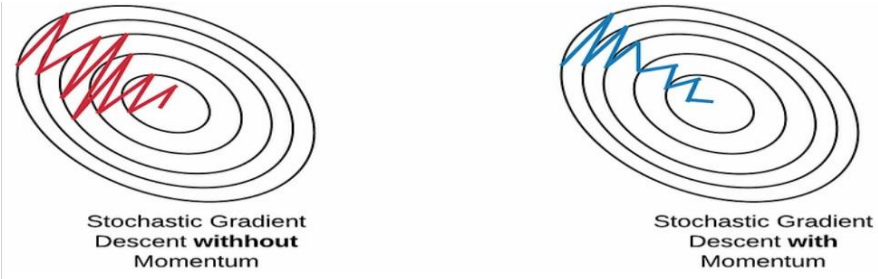
❖ *Momentum*

Momentum was invented for reducing high variance in SGD and softens the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction. One more hyperparameter is used in this method known as momentum symbolized by 'γ'.

$$V(t)=\gamma V(t-1)+\alpha.\nabla J(\theta)$$

Now, the weights are updated by $\theta=\theta-V(t)$.

The momentum term γ is usually set to 0.9 or a similar value.



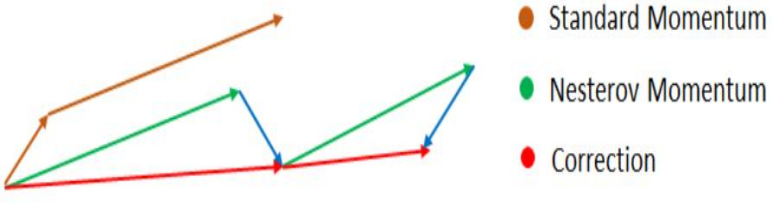
Advantages:

- Reduces the oscillations and high variance of the parameters.
- Converges faster than gradient descent.

Disadvantages:

- One more hyper-parameter is added which needs to be selected manually and accurately.

❖ *Nesterov Accelerated Gradient*



Momentum may be a good method but if the momentum is too high the algorithm may miss the local minima and may continue to rise up. So, to resolve this issue the NAG algorithm was developed. It is

a look ahead method. We know we'll be using **γV(t−1)** for modifying the weights so, **θ−γV(t−1)** approximately tells us the future location. Now, we'll calculate the cost based on this future parameter rather than the current one.

V(t)=γV(t−1)+α. ∇J(θ−γV(t−1)) and then update the parameters using **θ=θ−V(t).**

Advantages:

- Does not miss the local minima.
- Slows if minima's are occurring.

Disadvantages:

- Still, the hyperparameter needs to be selected manually.

❖ *Adagrad*

One of the disadvantages of all the optimizers explained is that the learning rate is constant for all parameters and for each cycle. This optimizer changes the learning rate. It changes the learning rate **'η'** for each parameter and at every time step **'t'.** It's a type second order optimization algorithm. It works on the derivative of an error function.

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}),$$

A derivative of loss function for given parameters at a given time t.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

Update parameters for given input **i** and at time/iteration **t**

η is a learning rate which is modified for given parameter **θ(i)** at a given time based on previous gradients calculated for given parameter **θ(i).**

We store the sum of the squares of the gradients w.r.t. **θ(i)** up to time step **t**, while **ϵ** is a smoothing term that avoids division by zero (usually on the order of 1e−8). Interestingly, without the square root operation, the algorithm performs much worse.

It makes big updates for less frequent parameters and a small step for frequent parameters.

Advantages:

- Learning rate changes for each training parameter.

- Don't need to manually tune the learning rate.
- Able to train on sparse data.

Disadvantages:

- Computationally expensive as a need to calculate the second order derivative.
- The learning rate is always decreasing results in slow training.

❖ *AdaDelta*

It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previously squared gradients, ***Adadelta*** limits the window of accumulated past gradients to some fixed size **w**. In this exponentially moving average is used rather than the sum of all the gradients.

$$\mathbf{E[g^2](t)=\gamma.E[g^2](t-1)+(1-\gamma).g^2(t)}$$

We set **γ** to a similar value as the momentum term, around 0.9.

$$\mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1-\gamma) g_t^2,$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} \cdot g_t.$$

Advantages:

- Now the learning rate does not decay and the training does not stop.

Disadvantages:

- Computationally expensive.

❖ *Adam*

Adam (Adaptive Moment Estimation) works with momentums of first and second order. The intuition behind the Adam is that we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search. In addition to storing an exponentially decaying average of past squared gradients like **AdaDelta**, ***Adam*** also keeps an exponentially decaying average of past gradients **M(t).**

M(t) and V(t) are values of the first moment which is the ***Mean*** and the second moment which is the ***uncentered variance*** of the gradients respectively.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Here, we are taking mean of **M(t)** and **V(t)** so that **E[m(t)]** can be equal to **E[g(t)]** where, **E[f(x)]** is an expected value of **f(x)**.

To update the parameter:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

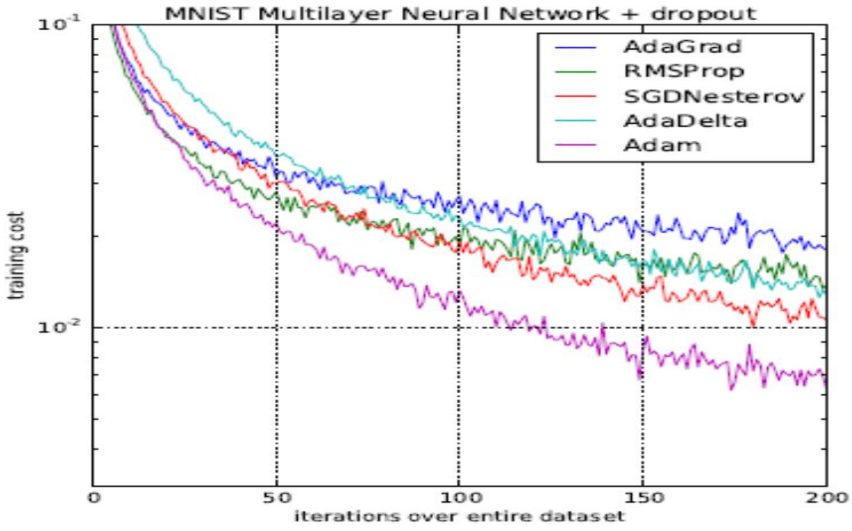
Advantages:

- The method is too fast and converges rapidly.
- Rectifies vanishing learning rate, high variance.

Disadvantages:

- Computationally costly.

➢ Comparison



As you can observe, the training cost in the case of Adam is the least.

Now observe the animation at the beginning of this article and consider the following points:

It is observed that the SGD algorithm (red) is stuck at a saddle point. So SGD algorithm can only be used for shallow networks.

All the other algorithms except SGD finally converges one after the other, AdaDelta being the fastest followed by momentum algorithms.

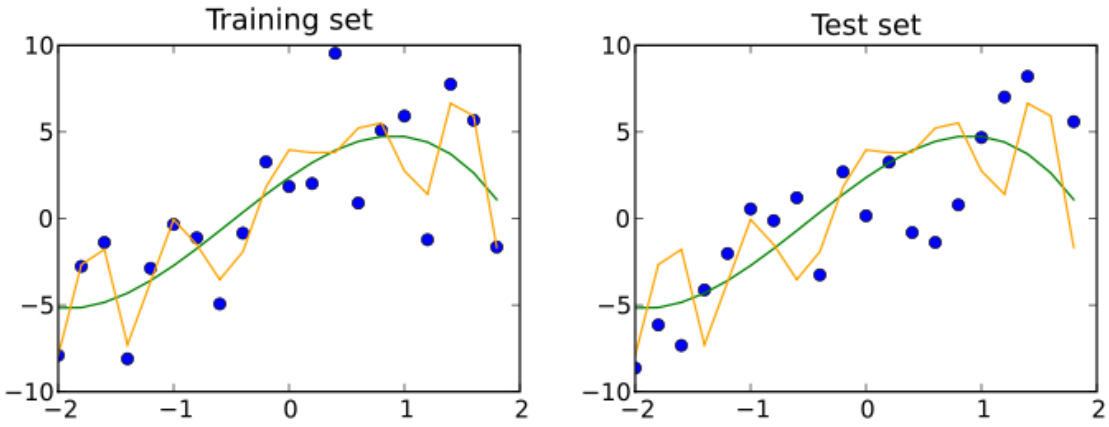
AdaGrad and AdaDelta algorithm can be used for sparse data.

Momentum and NAG work well for most cases but is slower.

Animation for Adam is not available but from the plot above it is observed that it is the fastest algorithm to converge to minima.

Adam is considered the best algorithm amongst all the algorithms discussed above.

9. Datasets



❖ *Training dataset*

A training dataset is a dataset of examples used during the learning process. This set includes two parts: input and output. Each of the parameters in the input is corresponding to a specified label in the output set.

A supervised learning algorithm looks at the training dataset to determine the optimal combinations of variables that will generate a good predictive model. The goal is to produce a trained (fitted) model that generalizes well to new, unknown data.

❖ *Validation dataset*

A validation dataset is a *dataset* of examples used to tune the architecture of a classifier, called the development set or the "dev set". The training dataset is used to train the different candidate classifiers, the validation dataset is used to compare their

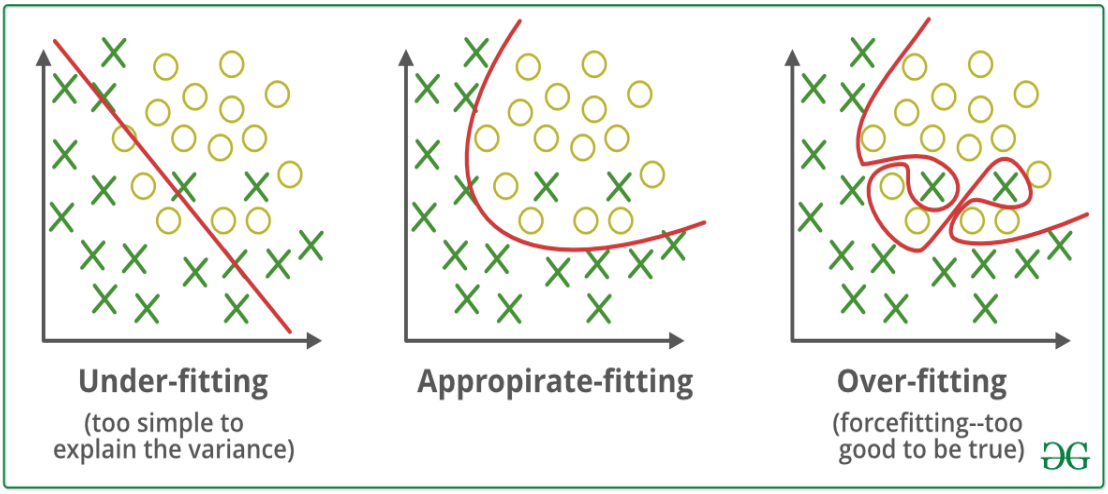
performances and decide which one to take and, finally, the test dataset is used to obtain the performance characteristics such as accuracy, sensitivity, specificity, F-measure, and so on. The validation dataset is training data used for testing, but neither as part of the low-level training nor as part of the final testing.

- ***Test dataset***

A test dataset is a dataset that is independent of the training dataset, but that follows the same probability distribution as the training dataset. If a model fit to the training dataset also fits the test dataset well, minimal overfitting has taken place .A better fitting of the training dataset as opposed to the test dataset usually points to overfitting.

10. Overfitting and Underfitting

Let us consider that we are designing a machine learning model. Suppose we want to check how well our machine learning model learns and generalizes to the new data. For that we have overfitting and underfitting, which are majorly responsible for the poor performances of the machine learning algorithms.



- ***Underfitting:***

A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data. Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough. It usually happens when we have less data to build an accurate model and also when we try to build a linear model with a non-linear data. In such cases the rules of the machine learning model

are too easy and flexible to be applied on such minimal data and therefore the model will probably make a lot of wrong predictions

- *Eliminating Underfitting*
  - Increase the size or number of parameters in the ML model.
  - Increase the complexity or type of the model.
  - Increasing the training time until cost function in ML is minimised.

❖ *Overfitting:*

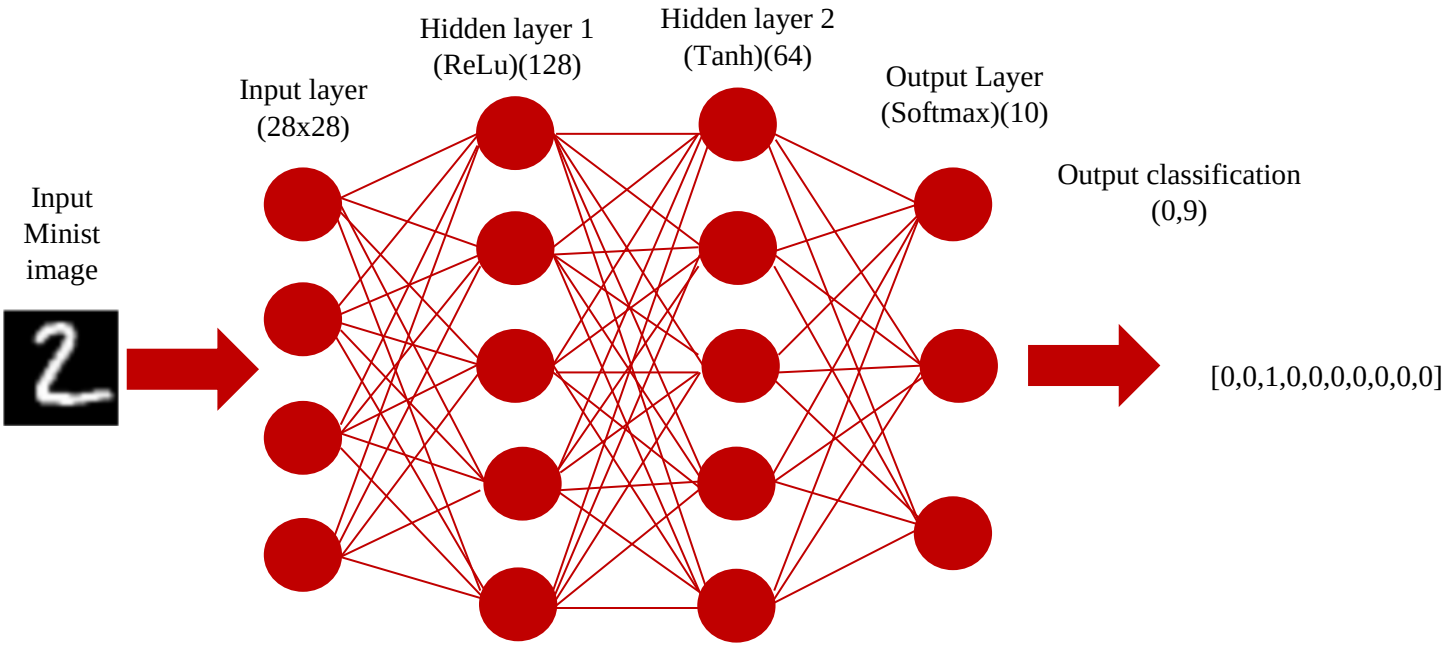
A statistical model is said to be overfitted, when we train it with a lot of data *(just like fitting ourselves in oversized pants!)*. When a model gets trained with so much of data, it starts learning from the noise and inaccurate data entries in our data set. Then the model does not categorize the data correctly, because of too many details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models. A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

- Techniques to reduce overfitting :
  - Increase training data.
  - Reduce model complexity.
  - Early stopping during the training phase (have an eye over the loss over the training period as soon as loss begins to increase stop training).
  - Ridge Regularization and Lasso Regularization
  - Use dropout for neural networks to tackle overfitting.

II. Performance

1. Model Architecture

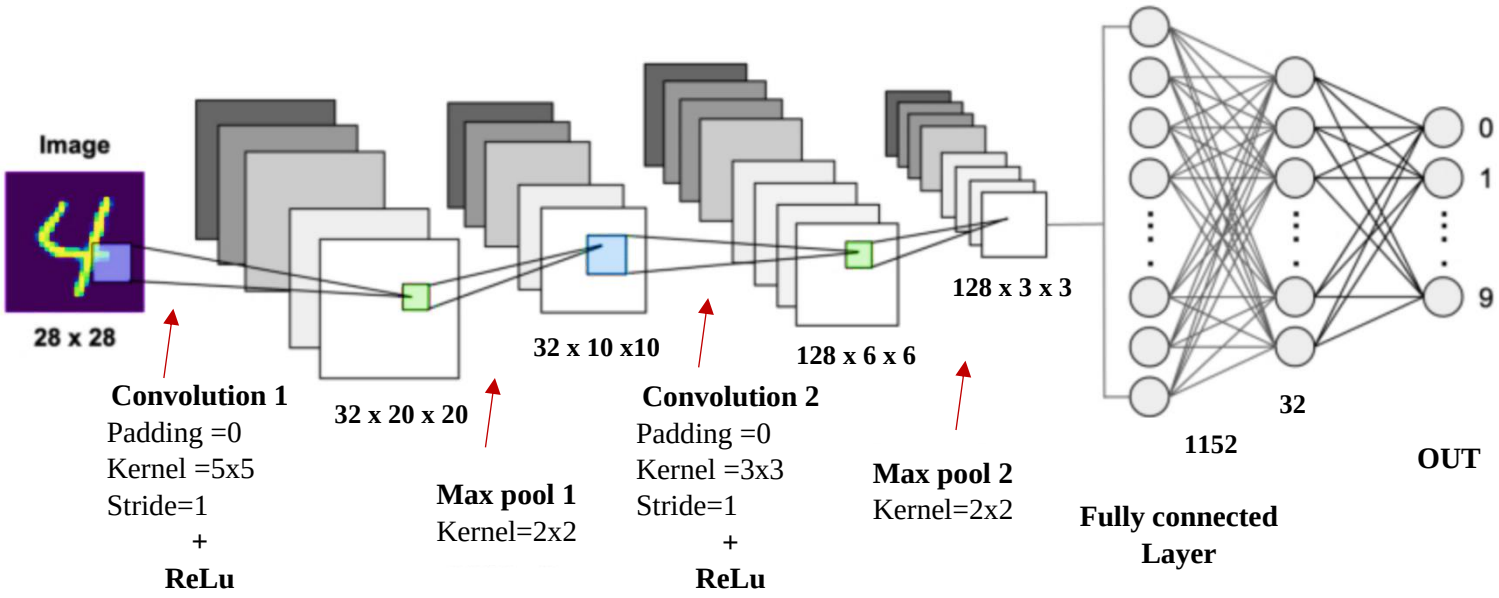
❖ *Fully-connected Model*



A fully connected neural network consists of a series of fully connected layers that connect every neuron in one layer to every neuron in the other layer. A *neural network* is simply a function that fits some data, typically called *neurons*. Each neuron has some number of weighted inputs. These weighted inputs are summed together (a linear combination) then passed through an activation function to get the unit's output.

We designed the model network with 4 layers: Input layer, Hidden layer 1, Hidden layer 2 and the Output layer. The Input layer takes the image of size 28x28 and forwards through 2 Hidden layers with ReLu and Tanh activation function. The log-softmax produces a probability close to 0 and 1. Finally, the model classifies which label number the input image should be chosen for.

❖ *Convolutional Neural Network*



The input image will go through 2 convolutional layers with ReLu activation function, 2 max-pooling layers and 1 fully-connected layer.

The first convolution layer takes in a channel of dimension 1 since the images are grayscaled. The kernel size is chosen to be of size 5x5 with stride of 1..This layer will do the convolutional process with ReLu twice , which has 1st out-channel set to 16 and the 2nd one is 32. Then it moves through a max-pooling layer with kernel size of 2. This downs the feature maps to dimension of 32x10x10.

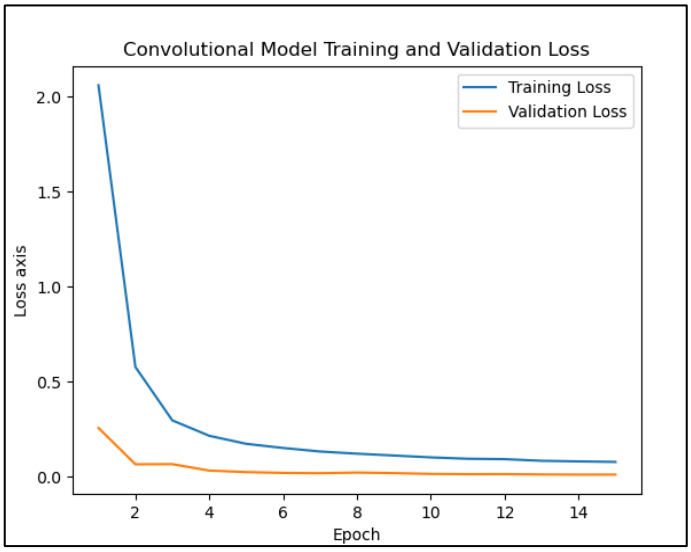
The second convolution layer is set 32 for input channel, 64 for the output channel, 3 for kernel size with stride 1. The convolutional process is done twice, just like the first convolutional layer, with out-channel of 64 for the first and 128 for the second. After that, the dimension will be downed to 128x3x3 through max-pooling layer.

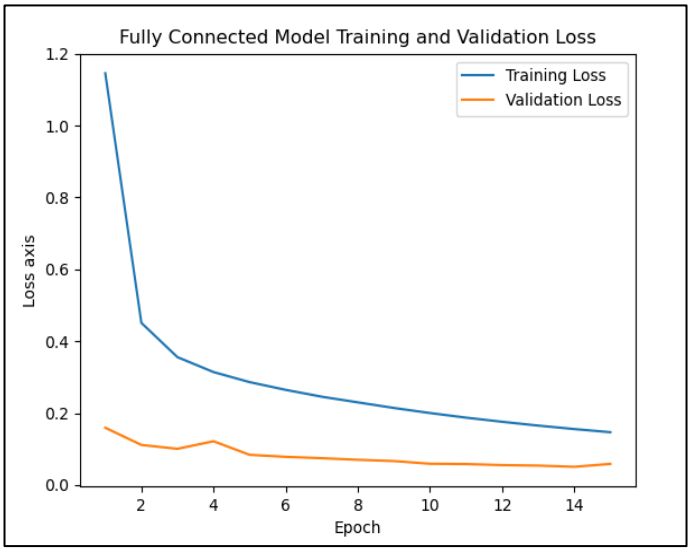
Finally, fully connected layers are used. We will pass a flattened version of the feature maps to the first fully connected layer. Then, each feature will go under the Log-Softmax function to be classified which number it belongs to.

2. Evaluating Model

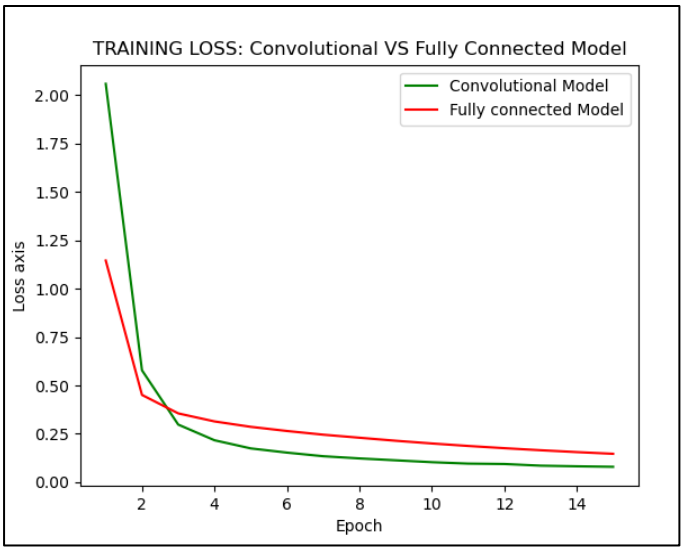
We use the Negative Log-likelihood Loss function to examine the training process.

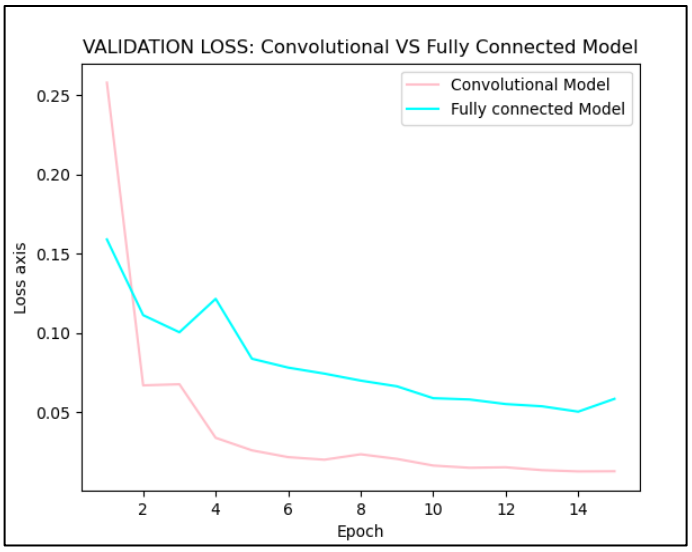
The negative log-likelihood is very useful if we combine it with the behavior of softmax to evaluate how happiness (correct) our model can perform. The model becomes unhappy at smaller value, when the network assigns low confidence at the correct class, which can reach infinite. But when the network assigns high confidence at the correct class, out model becomes less unhappy at larger values.



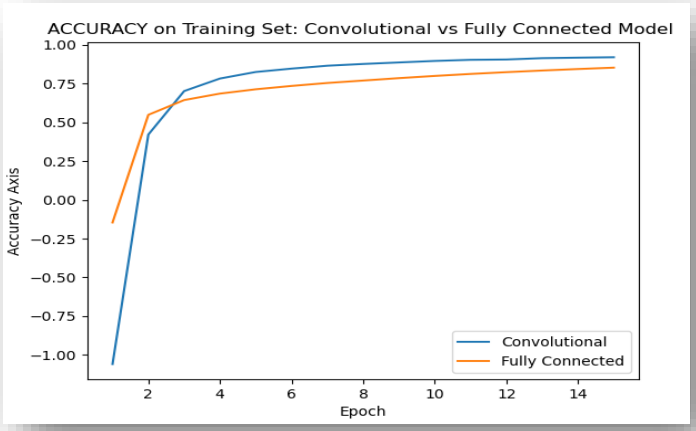


The above graphs give information about how change in loss value during training time.Training line in both graphs decrease rapidly from the beginning to epoch 2 and then slightly decline. The validation sets has the line moving down gradually to 0.





The 3rd and the 4th graph show the relationship between Training loss and Validation loss. At the begginning, the Convolutional Model has higher loss value than the Fully-connected one. But from epoch 3, its line suddenly falls and stays below the Fully's line. Apparently, the Convolutional Model has validation value smaller than the Fully-connected Model.

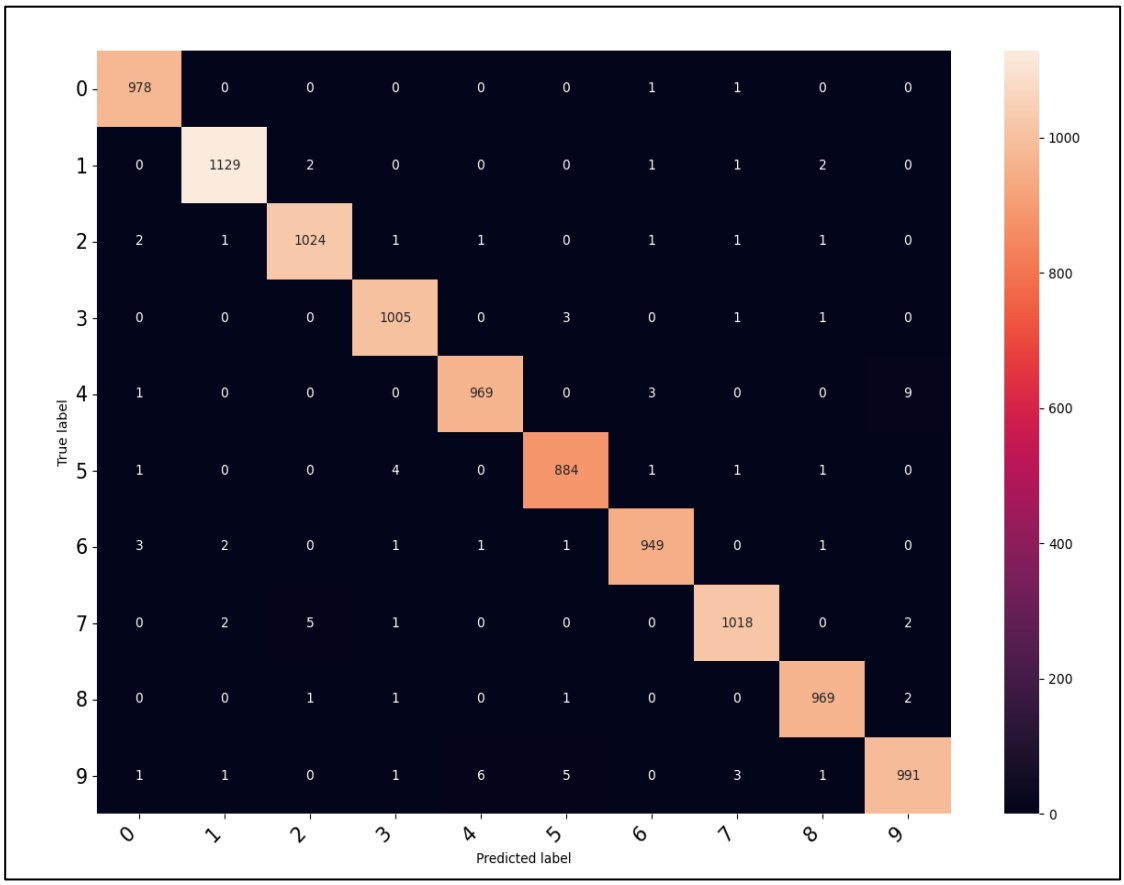


And finally, we plotted out the accuracy graph, which shows that Convolutional Model has higher exact ratio in prediction.

3. Model Result

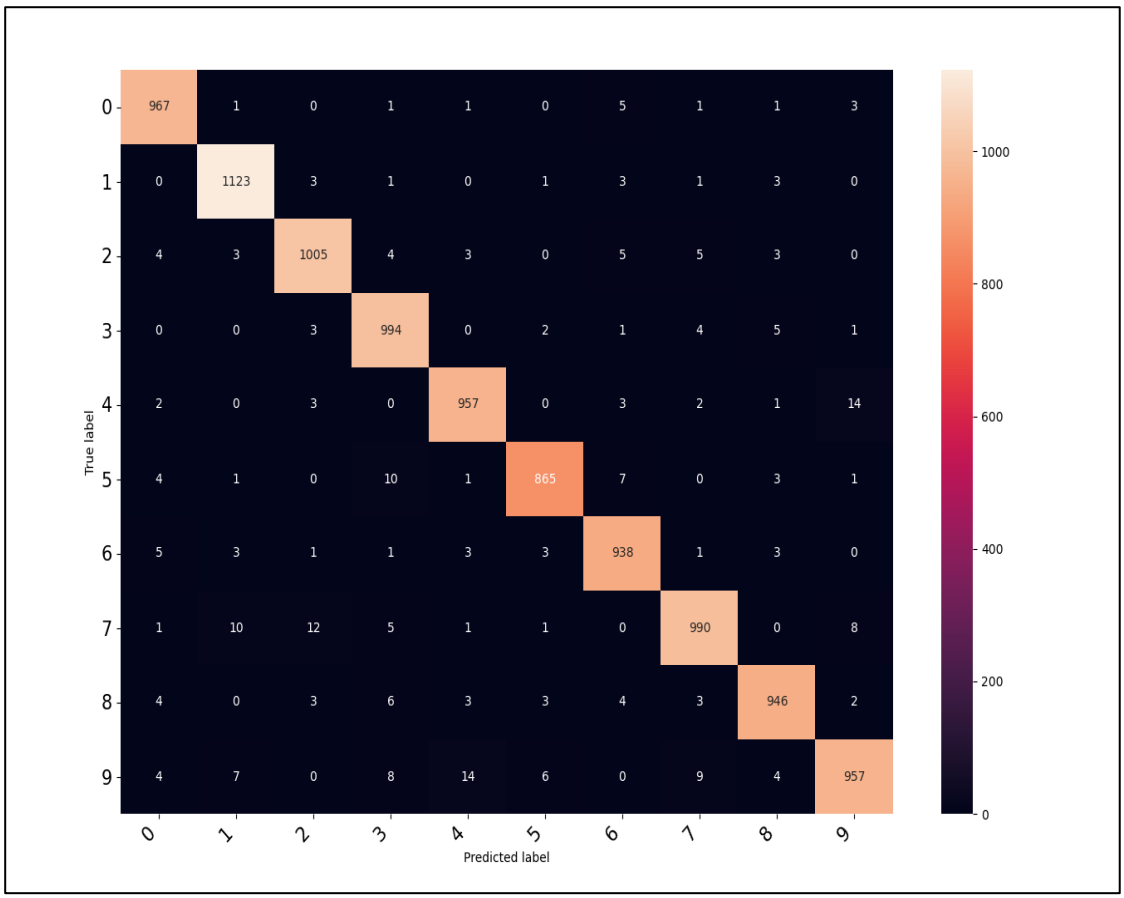
❖ ***Metric evaluating***

Metrics	Convolutional	Fully connected
Precision	0.9972605486449165	0.9877045522473799
Recall	0.99726	0.9877
F1 Score	0.9972597825619844	0.987695176848938
Cohens Kappa	0.9969546769257095	0.9863286173017671



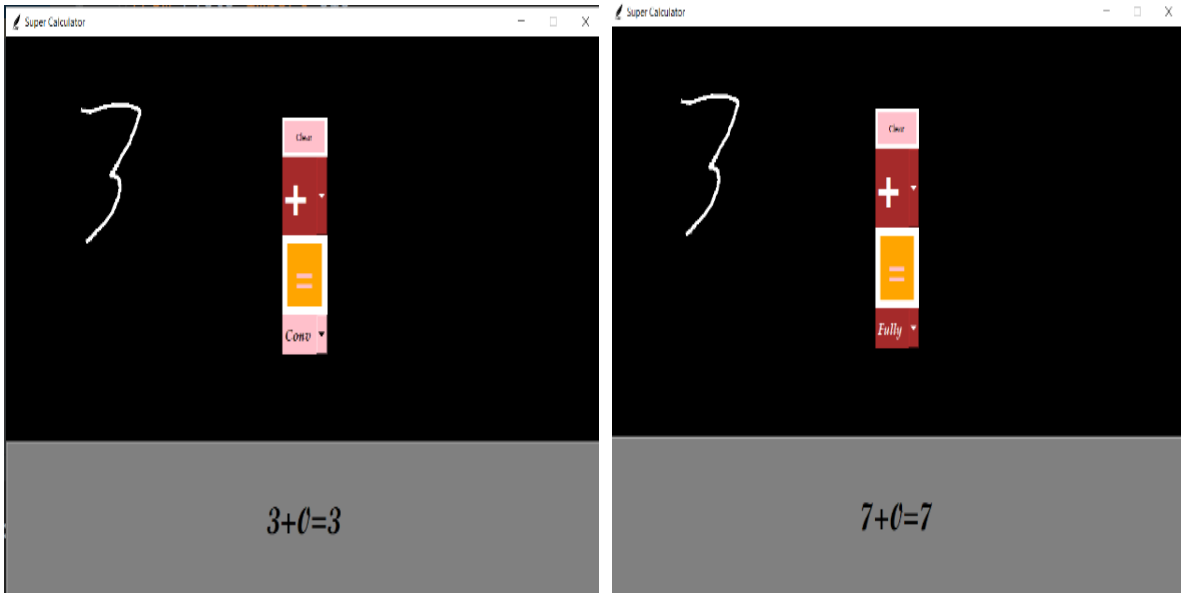
Confusion Matrix

Convolutional

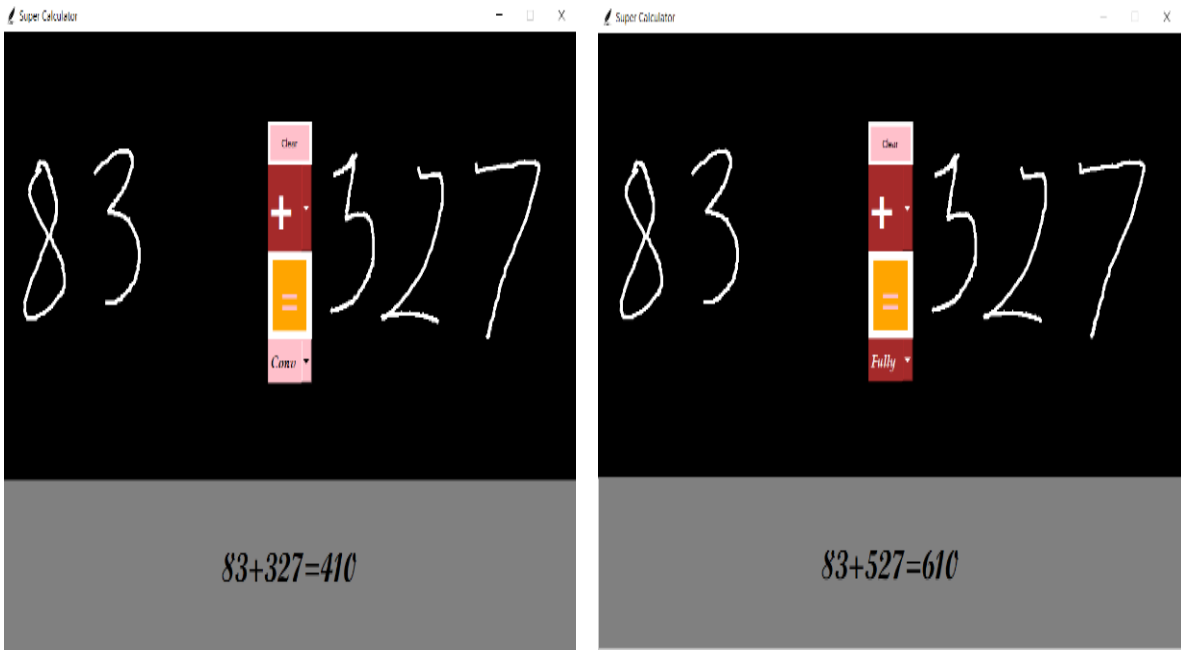


Fully connected

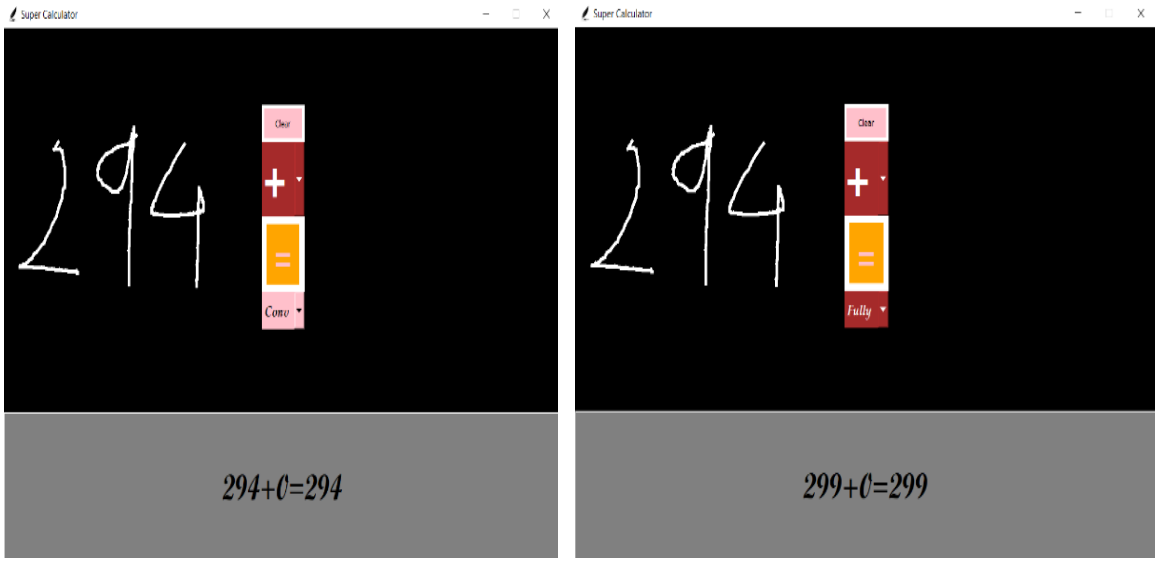
❖ *Some different results*



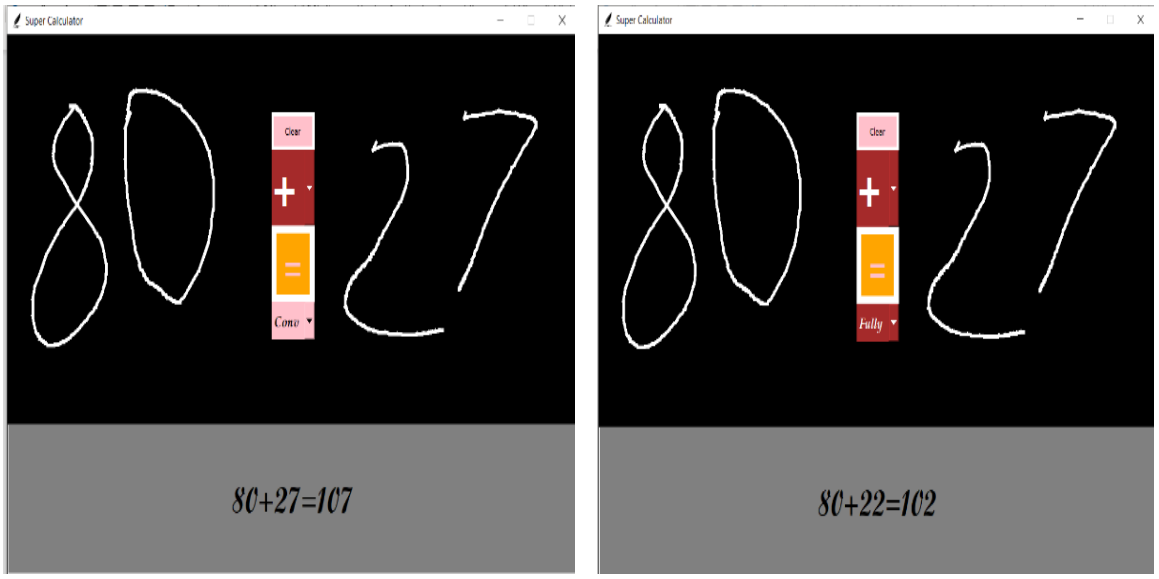
The number is 3, but it's top looks like 7 and when resized to 28 x 28 it looks more like 7, because the middle is thinker and the body more like a straight line. That's why fully connected predicted wrong in this case.



The body of number 3 is now look like the body of number 5 so FC missed it. However, CNN summarize the image features and yield a more concise feature maps.



In CNN, it is typically true of images that pixels in close proximity are more related with each other than with pixels that have greater distance.



In CNN, pooling layers downscale the image. This is possible because we retain throughout the network,features that are organized spatially like an image, and thus downscaling them makes sense as reducing the size of the image.

CNN Model (pictures on the left side) have more correct predictions than the Fully-connected one (on the right side)

CNN automatically detects the important features without any human supervision. Given many pictures of handwritten digits, it learns distinctive features for 10 classes (0-9) by itself. CNN exploit the strong spatially local correlation present in images.

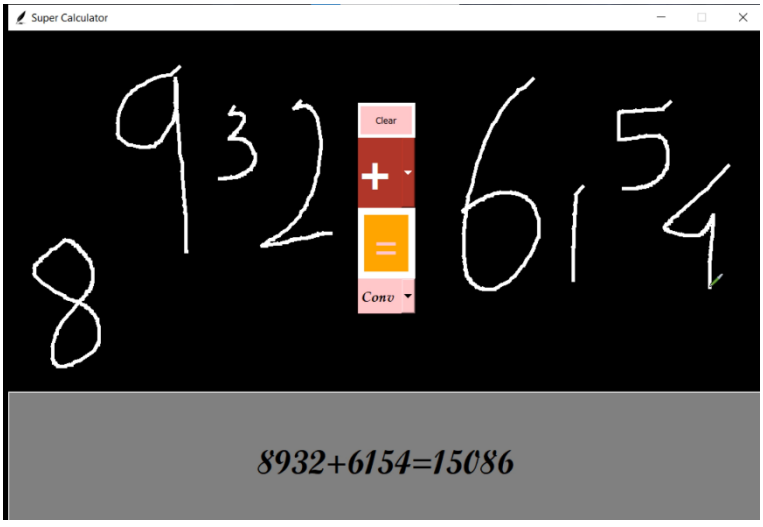
Fully connected does not take into account the spatial structure of data, treating input pixels which are far apart in the same way as pixels that are close together. This ignores locality of reference in data with a grid-topology like images, both computationally and semantically. Image recognition that are dominated by spatially local input patterns, therefore full connectivity of neurons is wasteful for purposes.

III.Real-life Model

Everybody needs a calculator. Because of this, we build this application to solve some simple calculation and it is actually pretty easy to use. Most people find it easier to write directly on the screen than type on the keyboard, and this app perfectly meet this expectation. Calculating app is used when we need to figure out stuff like how much to pay at market or when going shopping, or students literally may have them for school. This is just a simple state of the calculatio app. In the future, we hope to improve this app for handling complex operation , such as sin,cos, tan, function, ….

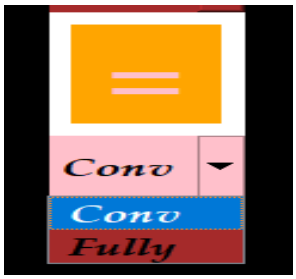
About the SUPER CALCULATOR App:

- Super Calculator allows users to write integer number on the left and right side of the mathematical operators (+,-,*,/).



- User can write numbers anywhere in the black board on both sides of the operator, but NUMBER must be separated from each other.
- The operation will be performed continuously without pressing the '=' sign.
- Results of identification numbers and handling the operations will be shown at the gray area.
- If no number is written , it will be default to 0

- There are 2 Models : CNN (recommended as higher accuracy) and Fully-connected.



➢ The video of this app :

https://www.youtube.com/watch?v=kqHEnRPHtVU&ab_channel=Minh%C4%90%E1%BB%A9c

Reference

https://wiki.pathmind.com/accuracy-precision-recall-f1

https://blog.floydhub.com/a-pirates-guide-to-accuracy-precision-recall-and-other-scores/#accuracy

https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234

https://machinelearningcoban.com/2017/08/31/evaluation/#-precision-va-recall

https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

https://builtin.com/data-science/gradient-descent

https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1

https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a

https://www.pluralsight.com/guides/building-your-first-pytorch-solution?fbclid=IwAR1dfqoGi6dZzCieVi6xU_6oaj0ARdnNMhWY52aXrvHycukYFgAE8Mzo5jQ

https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234

https://medium.com/@zeeshanmulla/cost-activation-loss-function-neural-network-deep-learning-what-are-these-91167825a4de

https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

https://www.kdnuggets.com/2020/12/optimization-algorithms-neural-networks.html#:~:text=Optimizers%20are%20algorithms%20or%20methods,problems%20by%20minimizing%20the%20function.

https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

https://analyticsindiamag.com/tackling-underfitting-and-overfitting-problems-in-data-science/

https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5#:~:text=A%20fully%20connected%20neural%20network%20consists%20of%20a%20series%20of,be%20made%20about%20the%20input.

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets

https://www.jeremyjordan.me/convolutional-neural-networks/