Debonil Ghosh (M21AIE225)

**Problem 2: Learning to implement Neural Network [40**

**points]**

## 2.2. Please go through the following blog to learn how to recognize handwritten digits using

## Neural Network. Here Neural Network is coded using PyTorch Library in Python.

## https://towardsdatascience.com/handwritten-digit-mnist-pytorch-977b5338e627

## Use above code and report your observation based on the following:(15 points)

## (i) Change loss function,

## (ii) Change in learning rate, and

## (iii) Change in Number of hidden layers

# Solution:

By updating given code, following observations have generated through python code.

#### (i) Change loss function:

#### Training with Loss Function = NLLLoss(), Learning Rate = 0.003, Hidden Layer = [128, 64]

#### Sequential(

#### (0): Linear(in\_features=784, out\_features=128, bias=True)

#### (1): ReLU()

#### (2): Linear(in\_features=128, out\_features=64, bias=True)

#### (3): ReLU()

#### (4): Linear(in\_features=64, out\_features=10, bias=True)

#### (5): LogSoftmax(dim=1)

#### )

#### Training Time (in minutes) = 2.5658482789993284

#### Overall Accuracy Score: 0.975

#### Classwise Accuracy Score: [0.9964 0.9973 0.9952 0.9943 0.9948 0.9947 0.9954 0.9945 0.9939 0.9929]

#### 

#### Confusion Matrix for NLLLoss

#### Training with Loss Function = MSELoss(), Learning Rate = 0.003, Hidden Layer = [128, 64]

#### Sequential(

#### (0): Linear(in\_features=784, out\_features=128, bias=True)

#### (1): ReLU()

#### (2): Linear(in\_features=128, out\_features=64, bias=True)

#### (3): ReLU()

#### (4): Linear(in\_features=64, out\_features=10, bias=True)

#### (5): LogSoftmax(dim=1)

#### )

#### Training Time (in minutes) = 7.808586394786834

#### Overall Accuracy Score: 0.938

#### Classwise Accuracy Score: [0.9916 0.9944 0.9846 0.9873 0.9871 0.9863 0.9899 0.9874 0.9839 0.9833]

#### 

#### Training with Loss Function = KLDivLoss(), Learning Rate = 0.003, Hidden Layer = [128, 64]

#### Sequential(

#### (0): Linear(in\_features=784, out\_features=128, bias=True)

#### (1): ReLU()

#### (2): Linear(in\_features=128, out\_features=64, bias=True)

#### (3): ReLU()

#### (4): Linear(in\_features=64, out\_features=10, bias=True)

#### (5): LogSoftmax(dim=1)

#### )

#### Training Time (in minutes) = 3.6359249035517376

#### Overall Accuracy Score: 0.928

#### Classwise Accuracy Score: [0.992 0.9928 0.9832 0.9835 0.986 0.9811 0.9886 0.9867 0.9798 0.9831]

#### 

#### Training with Loss Function = CrossEntropyLoss(), Learning Rate = 0.003, Hidden Layer = [128, 64]

#### Sequential(

#### (0): Linear(in\_features=784, out\_features=128, bias=True)

#### (1): ReLU()

#### (2): Linear(in\_features=128, out\_features=64, bias=True)

#### (3): ReLU()

#### (4): Linear(in\_features=64, out\_features=10, bias=True)

#### (5): LogSoftmax(dim=1)

#### )

#### Training Time (in minutes) = 2.653095289071401

#### Overall Accuracy Score: 0.975

#### Classwise Accuracy Score: [0.9966 0.9973 0.9949 0.9948 0.9943 0.9955 0.9957 0.9936 0.994 0.9933]

#### 

## (ii) Change in learning rate:

Training with Loss Function = NLLLoss(), **Learning Rate = 0.5,** Hidden Layer = [128, 64]

Sequential(

(0): Linear(in\_features=784, out\_features=128, bias=True)

(1): ReLU()

(2): Linear(in\_features=128, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=10, bias=True)

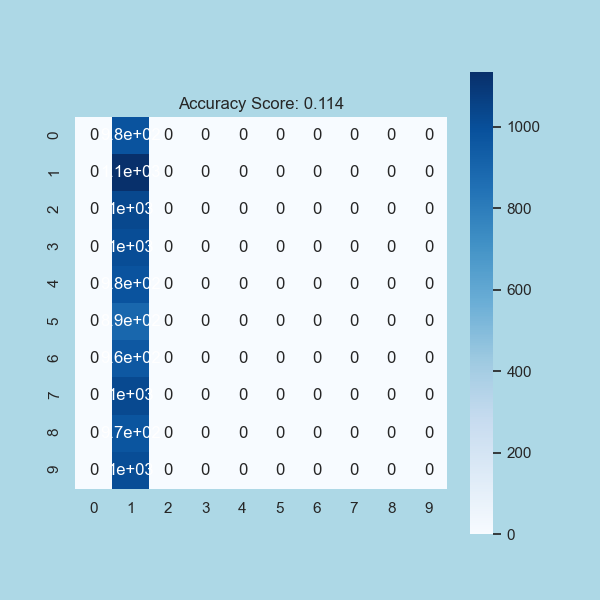
(5): LogSoftmax(dim=1)

)

Training Time (in minutes) = 2.7291388670603434

Overall Accuracy Score: **0.114**

Classwise Accuracy Score: [0.902 0.1135 0.8968 0.899 0.9018 0.9108 0.9042 0.8972 0.9026 0.8991]



Training with Loss Function = NLLLoss(), **Learning Rate = 0.01,** Hidden Layer = [128, 64]

Sequential(

(0): Linear(in\_features=784, out\_features=128, bias=True)

(1): ReLU()

(2): Linear(in\_features=128, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=10, bias=True)

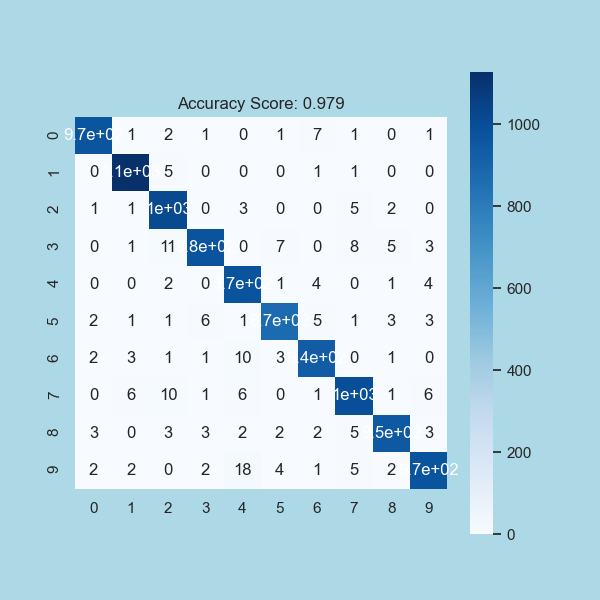
(5): LogSoftmax(dim=1)

)

Training Time (in minutes) = 3.002375260988871

Overall Accuracy Score: **0.979**

Classwise Accuracy Score: [0.9976 0.9978 0.9953 0.9951 0.9948 0.9959 0.9958 0.9943 0.9962 0.9944]



## (iii) Change in Number of hidden layers:

Training with Loss Function = NLLLoss(), Learning Rate = 0.01, **Hidden Layer = [128]**

**Sequential(**

**(0): Linear(in\_features=784, out\_features=128, bias=True)**

**(1): ReLU()**

**(2): Linear(in\_features=128, out\_features=10, bias=True)**

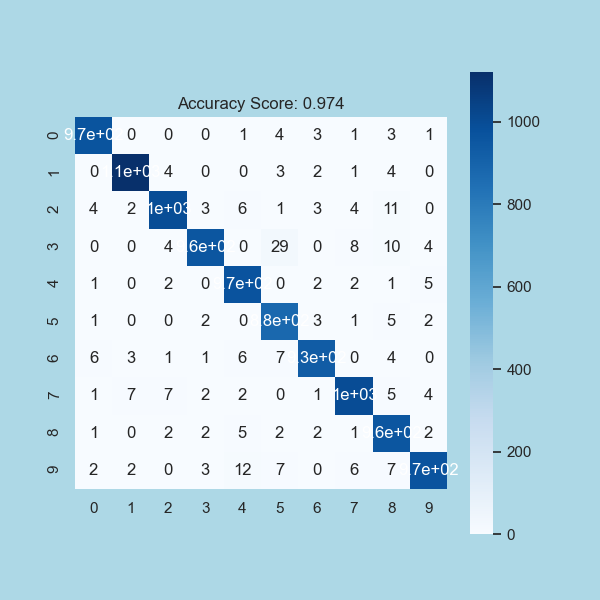
**(3): LogSoftmax(dim=1)**

**)**

Training Time (in minutes) = 5.416209638118744

Overall Accuracy Score: **0.974**

Classwise Accuracy Score: [0.9971 0.9972 0.9946 0.9932 0.9955 0.9933 0.9956 0.9947 0.9933 0.9943]



Training with Loss Function = NLLLoss(), Learning Rate = 0.01, **Hidden Layer = [16]**

**Sequential(**

**(0): Linear(in\_features=784, out\_features=16, bias=True)**

**(1): ReLU()**

**(2): Linear(in\_features=16, out\_features=10, bias=True)**

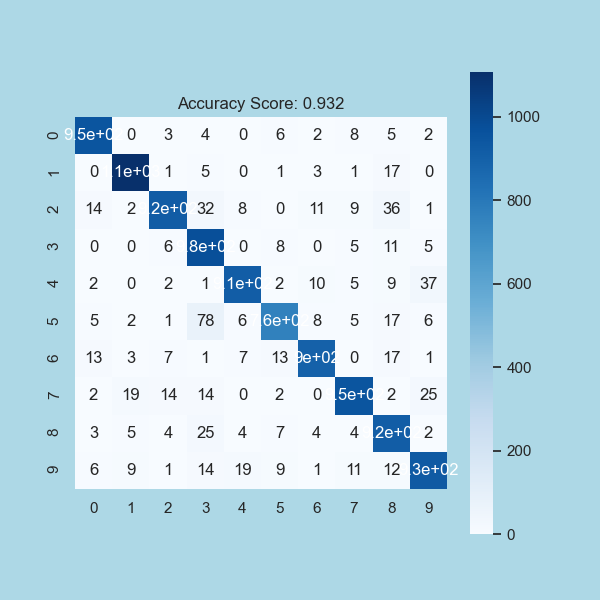
**(3): LogSoftmax(dim=1)**

**)**

Training Time (in minutes) = 2.390563261508942

Overall Accuracy Score: **0.932**

Classwise Accuracy Score: [0.9925 0.9932 0.9848 0.9791 0.9888 0.9824 0.9899 0.9874 0.9816 0.9839]



Training with Loss Function = NLLLoss(), Learning Rate = 0.01, **Hidden Layer = [256, 128]**

Sequential(

(0): Linear(in\_features=784, out\_features=256, bias=True)

(1): ReLU()

(2): Linear(in\_features=256, out\_features=128, bias=True)

(3): ReLU()

(4): Linear(in\_features=128, out\_features=10, bias=True)

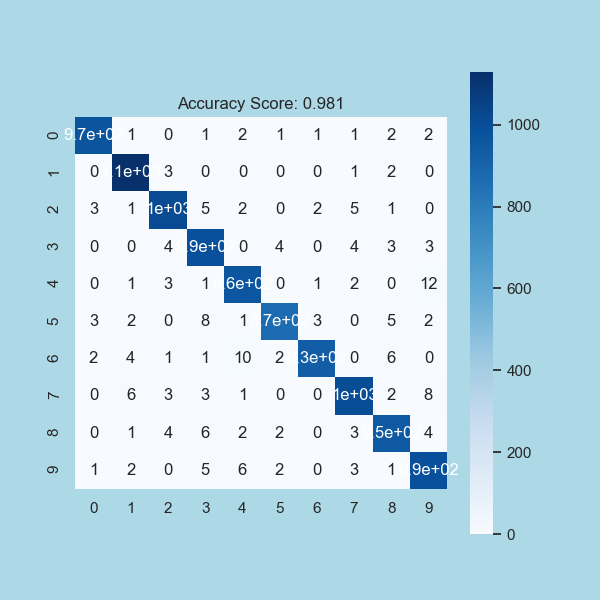
(5): LogSoftmax(dim=1)

)

Training Time (in minutes) = 4.351998011271159

Overall Accuracy Score: **0.981**

Classwise Accuracy Score: [0.998 0.9976 0.9963 0.9952 0.9956 0.9965 0.9967 0.9958 0.9956 0.9949]



Training with Loss Function = NLLLoss(), Learning Rate = 0.01, **Hidden Layer = [64, 16, 16, 16]**

Sequential(

(0): Linear(in\_features=784, out\_features=64, bias=True)

(1): ReLU()

(2): Linear(in\_features=64, out\_features=16, bias=True)

(3): ReLU()

(4): Linear(in\_features=16, out\_features=16, bias=True)

(5): ReLU()

(6): Linear(in\_features=16, out\_features=16, bias=True)

(7): ReLU()

(8): Linear(in\_features=16, out\_features=10, bias=True)

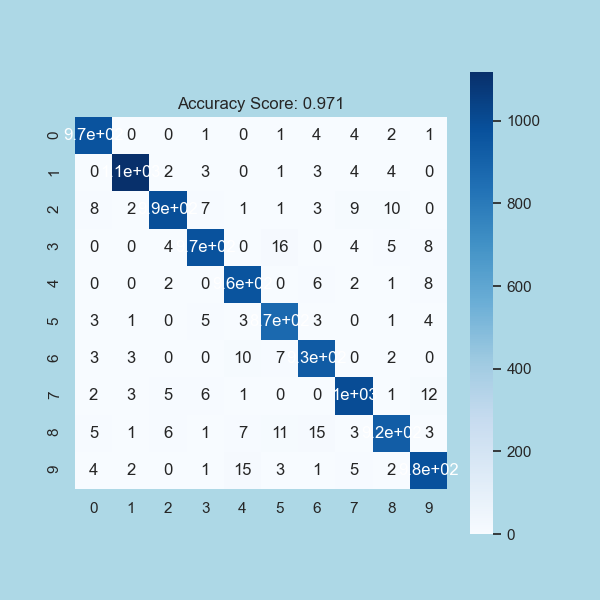
(9): LogSoftmax(dim=1)

)

Training Time (in minutes) = 7.081622072060903

Overall Accuracy Score: **0.971**

Classwise Accuracy Score: [0.9962 0.9971 0.994 0.9939 0.9944 0.994 0.994 0.9939 0.992 0.9931]



Training with Loss Function = NLLLoss(), Learning Rate = 0.01, **Hidden Layer = [256, 128, 64, 32]**

Sequential(

(0): Linear(in\_features=784, out\_features=256, bias=True)

(1): ReLU()

(2): Linear(in\_features=256, out\_features=128, bias=True)

(3): ReLU()

(4): Linear(in\_features=128, out\_features=64, bias=True)

(5): ReLU()

(6): Linear(in\_features=64, out\_features=32, bias=True)

(7): ReLU()

(8): Linear(in\_features=32, out\_features=10, bias=True)

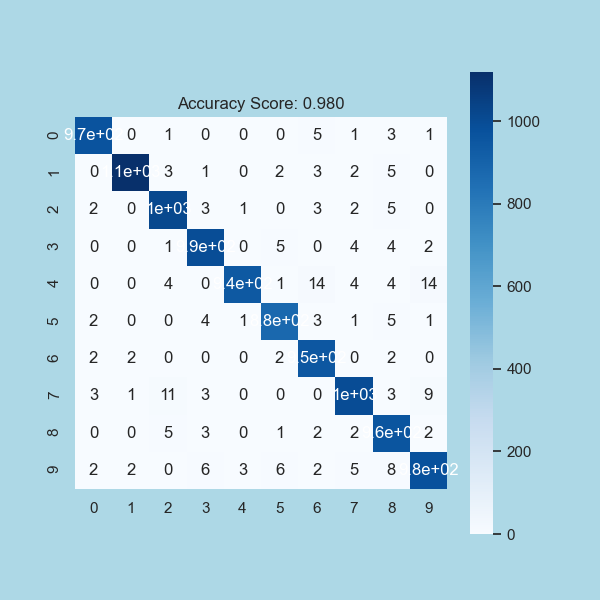
(9): LogSoftmax(dim=1)

)

Training Time (in minutes) = 7.422947756449381

Overall Accuracy Score: **0.980**

Classwise Accuracy Score: [0.9978 0.9979 0.9959 0.9964 0.9954 0.9966 0.996 0.9949 0.9946 0.9937]



2.3. Gurmukhi Handwritten Digit Classification: Gurmukhi is one of the popular

Indian scripts widely used in Indian state of Punjab. In this part of the assignment,

our goal is to develop a neural network solution for classifying Gurmukhi digits. We

provide you Handwritten Gurmukhi digit dataset here:

Dataset link

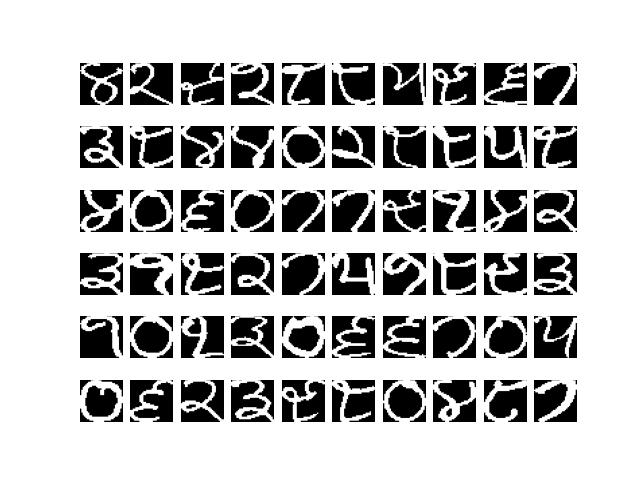
Modifying the code provided in 2, and develop a robust neural network to classify

the Gurmukhi digits. Higher performance on test set will have bonus point. Briefly

write your observation and submit your code so that we can evaluate your implementation at our end. (15 points)

# Solution:

#### Sample Data Images:



#### Model:

Sequential(

(0): Linear(in\_features=1024, out\_features=256, bias=True)

(1): ReLU()

(2): Linear(in\_features=256, out\_features=64, bias=True)

(3): ReLU()

(4): Linear(in\_features=64, out\_features=10, bias=True)

(5): LogSoftmax(dim=1)

)

Before backward pass:

None

After backward pass:

tensor([[ 3.8962e-04, 3.8962e-04, 3.8962e-04, ..., -1.3182e-03,

-5.4675e-04, -5.9323e-04],

[ 6.5492e-04, 6.5492e-04, 8.4178e-04, ..., -2.3123e-05,

2.2393e-04, 4.6235e-05],

[ 1.0681e-03, 1.0681e-03, 1.2085e-03, ..., -9.3416e-04,

-5.7025e-04, -3.3054e-04],

...,

[-1.1778e-03, -1.1778e-03, -1.1288e-03, ..., -1.1319e-04,

-2.0197e-04, 1.4466e-03],

[ 2.4794e-03, 2.3624e-03, 2.0342e-03, ..., -8.8682e-04,

9.1285e-04, 2.0330e-03],

[-5.5383e-04, -9.1174e-04, -1.0390e-03, ..., -2.2249e-03,

-1.9248e-03, -6.8608e-04]])

Epoch 0 - Training loss: 2.262930393218994 no of images 1000

Epoch 1 - Training loss: 2.079546920955181 no of images 1000

Epoch 2 - Training loss: 1.7204681560397148 no of images 1000

Epoch 3 - Training loss: 1.1766018122434616 no of images 1000

Epoch 4 - Training loss: 0.6834566779434681 no of images 1000

Epoch 5 - Training loss: 0.40679728984832764 no of images 1000

Epoch 6 - Training loss: 0.280624826438725 no of images 1000

Epoch 7 - Training loss: 0.21215492766350508 no of images 1000

Epoch 8 - Training loss: 0.17039840202778578 no of images 1000

Epoch 9 - Training loss: 0.1459644865244627 no of images 1000

Epoch 10 - Training loss: 0.12271467410027981 no of images 1000

Epoch 11 - Training loss: 0.10761574958451092 no of images 1000

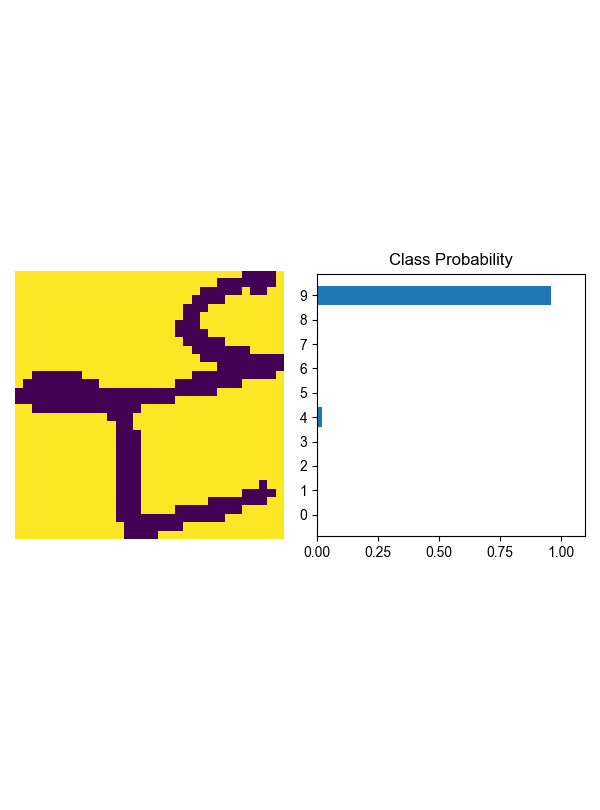
Epoch 12 - Training loss: 0.0930116605013609 no of images 1000

Epoch 13 - Training loss: 0.08325718808919191 no of images 1000

Epoch 14 - Training loss: 0.07367376610636711 no of images 1000

Training Time (in minutes) = 0.06675013303756713

Predicted Digit = 9



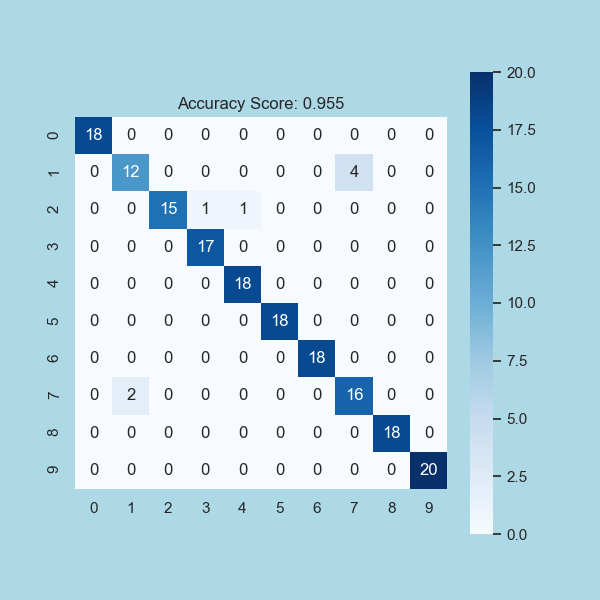
Number Of Images Tested = 178

Model Accuracy = 0.9550561797752809

Overall Accuracy Score: **0.955**

Classwise Accuracy Score: [1. 0.96629213 0.98876404 0.99438202 0.99438202 1.

1. 0.96629213 1. 1. ]



Confusion Matrix for Final prediction of Gurmukhi Image Classification

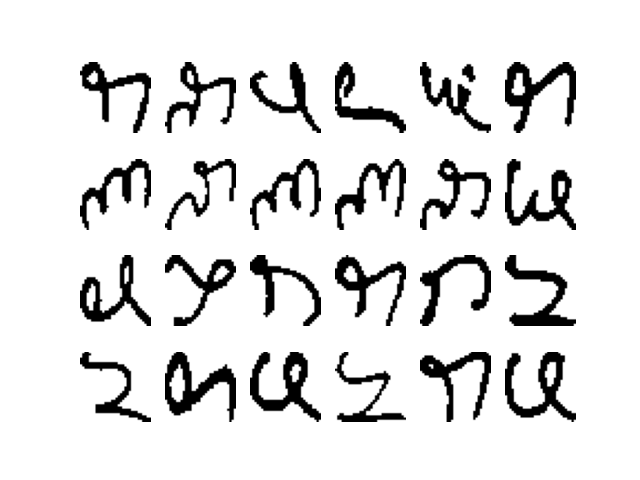
## **Problem 4: SVM [20 Points]**

## 1. Solve Subproblem 3 of Problem 2 (i.e. Gurmukhi Handwritten Digit Classification)

## using SVM. Use raw pixel features and linear, polynomial and RBF Kernel. Resource:

## https://scikit-learn.org/stable/modules/svm.html

# Solution:



Accuracy **Linear Kernel: 0.9719101123595506**

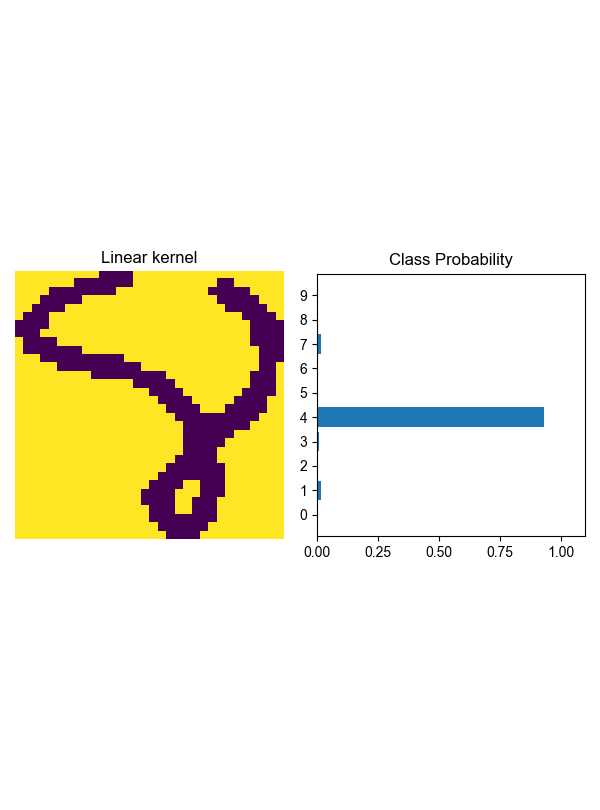
Accuracy **Polynomial Kernel: 0.9662921348314607**

Accuracy **Radial Basis Kernel: 0.20224719101123595**

Accuracy **Sigmoid Kernel: 0.9550561797752809**

Model = Linear kernel

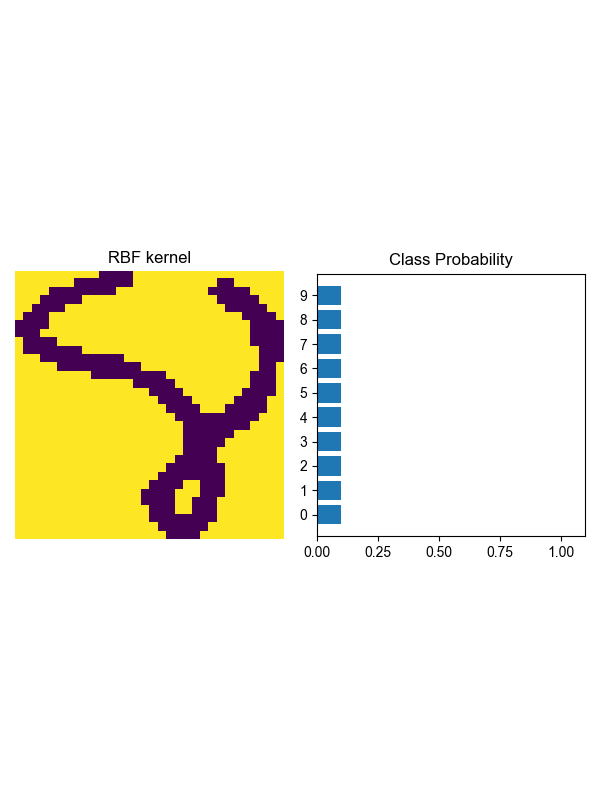
Predicted Digit = 4



- - - - -

Model = RBF kernel

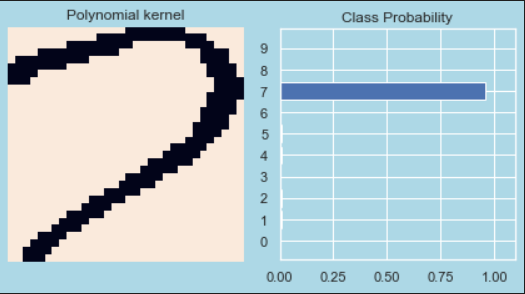
Predicted Digit = 0



- - - - -

Model = Polynomial kernel

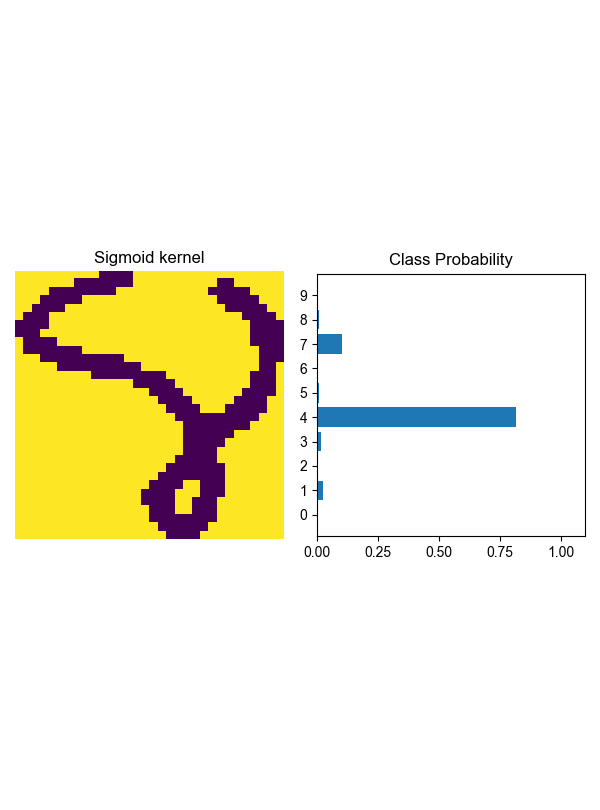
Predicted Digit = 4



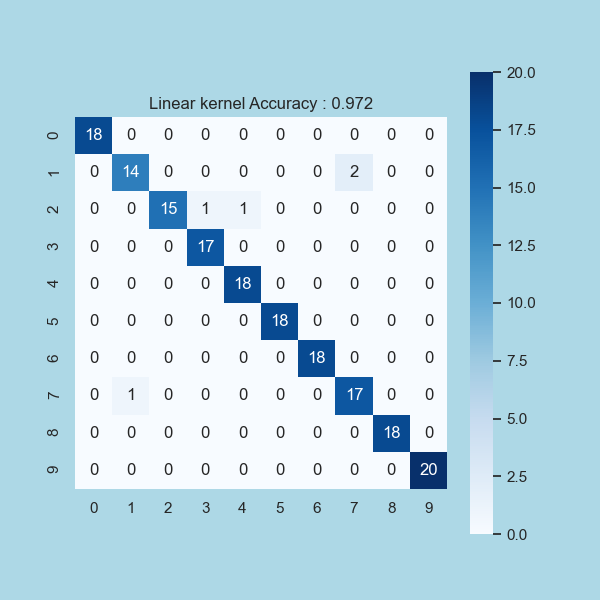
- - - - -

Model = Sigmoid kernel

Predicted Digit = 4



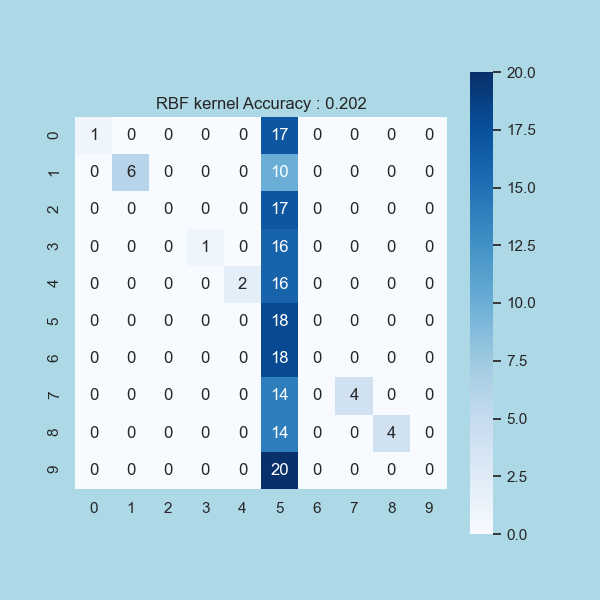
- - - - -



Overall Accuracy Score: 0.972

Classwise Accuracy Score: [1. 0.98314607 0.98876404 0.99438202 0.99438202 1.

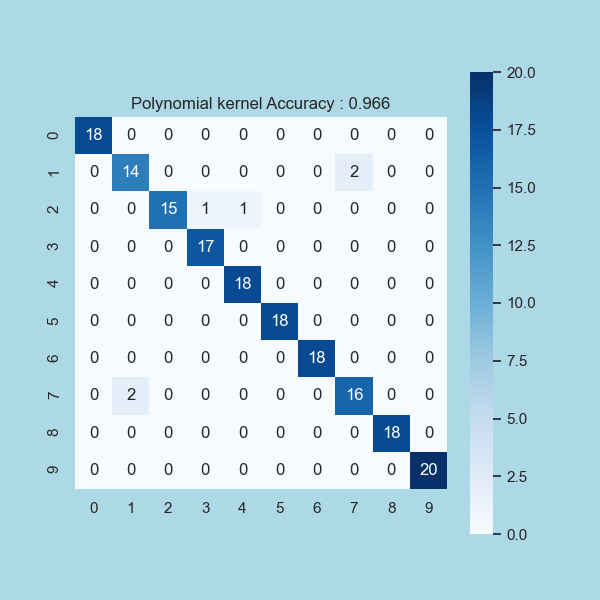
1. 0.98314607 1. 1. ]



Overall Accuracy Score: 0.202

Classwise Accuracy Score: [0.90449438 0.94382022 0.90449438 0.91011236 0.91011236 0.20224719

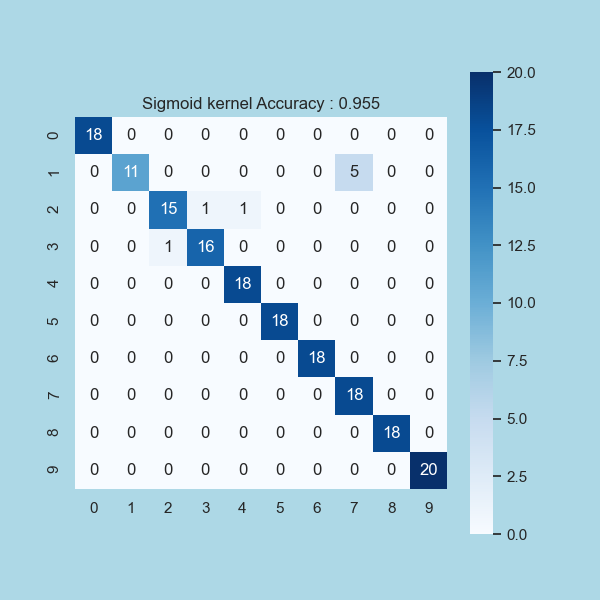
0.8988764 0.92134831 0.92134831 0.88764045]



Overall Accuracy Score: 0.966

Classwise Accuracy Score: [1. 0.97752809 0.98876404 0.99438202 0.99438202 1.

1. 0.97752809 1. 1. ]



Overall Accuracy Score: 0.955

Classwise Accuracy Score: [1. 0.97191011 0.98314607 0.98876404 0.99438202 1.

1. 0.97191011 1. 1. ]

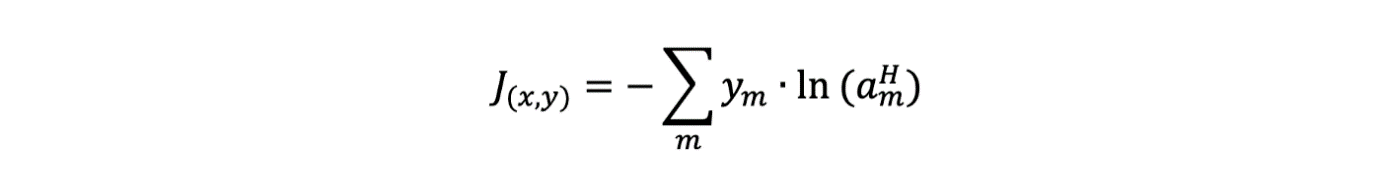
## Problem 2: Learning to implement Neural Network [40 points]

### 2.1. We have seen in class how backpropagation is used to compute gradients of cost functions with respect to weights. In class example we took Mean Square Error as cost function. Replace this cost function with cross-entropy loss and demonstrate backpropagation. Submit both handwritten explanation as well as modified code. Code link (10 points)

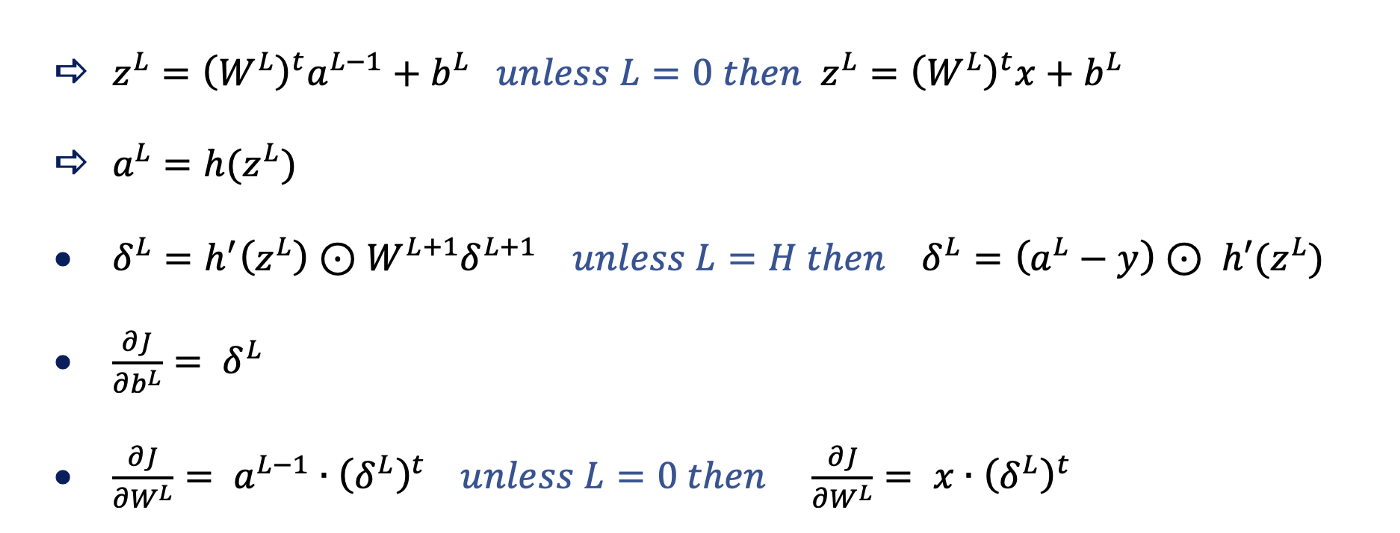
### Solution:

Cross-Entropy loss is a popular choice if the problem at hand is a classification problem, and in and of itself it can be classified into either categorical cross-entropy or multi-class cross-entropy (with binary cross-entropy being a special case of the former.)

Let’s start with categorical cross-entropy. For this loss function our y’s are one-hot encoded to denote the class our image (or whatever) belongs to. Thus for any x, y is of length equal to the number of classes and the last layer in our model has a neuron for each class. We use Softmax in our last layer to get the probability of x belonging to each of the classes. These probabilities sum to 1.



Categorical Cross-Entropy Given One Example. aᴴₘ is the mth neuron of the last layer (H)

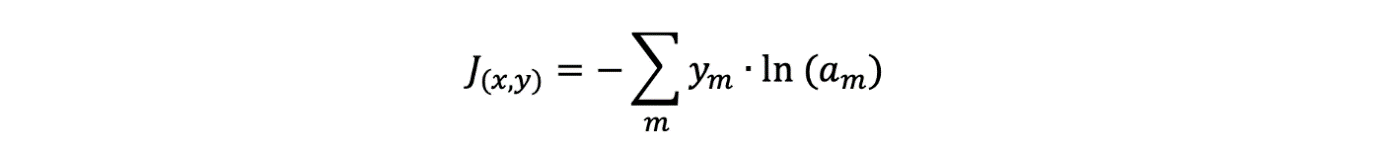


L=0 is the first hidden layer, L=H is the last layer. δ is ∂J/∂z

Note that the output (activations vector) for the last layer is aᴴ and in index notation we would write aᴴₙ to denote the nth neuron in the last layer. The same applies for the pre-activations vector zᴴ.

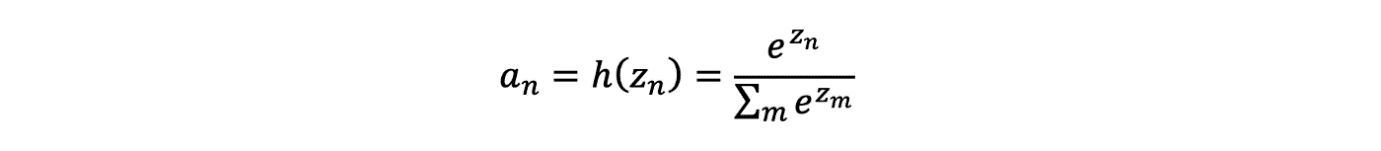
The only equation that we expect to change in the system above is that for δᴴ because we used the explicit formula for the loss function to find that. As a result, we’ll be only dealing with the **last layer** in the derivation so we might as well drop the superscript for now and keep in mind that we’re targeting the last layer whenever we write a, z or δ.

By this, the loss function is



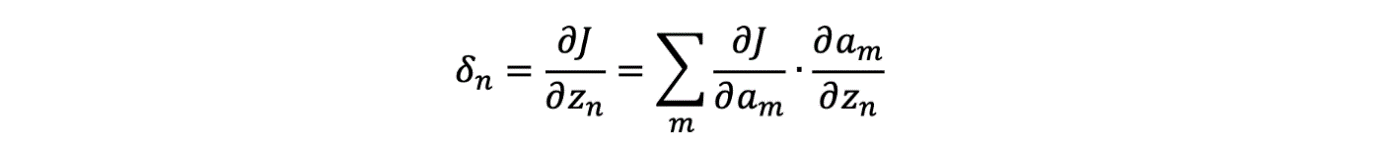
Categorical Cross-Entropy.

with the activation of the nth neuron in the last layer being



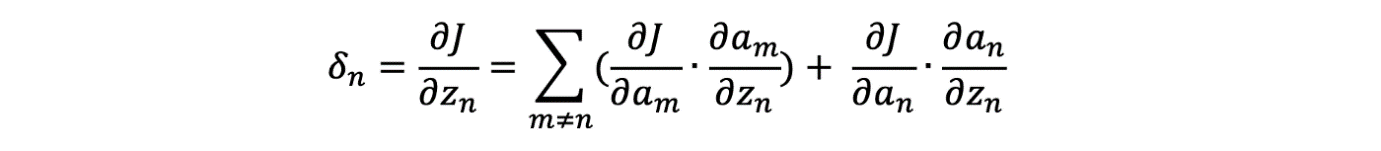
Softmax Activation. We’ll use this below many times. Keep it in mind.

Notice that the activation of the nth neuron depends on the pre-activations of all other neurons in the layer. This would’ve not been the case if the last layer involved Sigmoid or ReLU activations. On this account, to find δ for some neuron in the last layer we use the chain-rule by writing



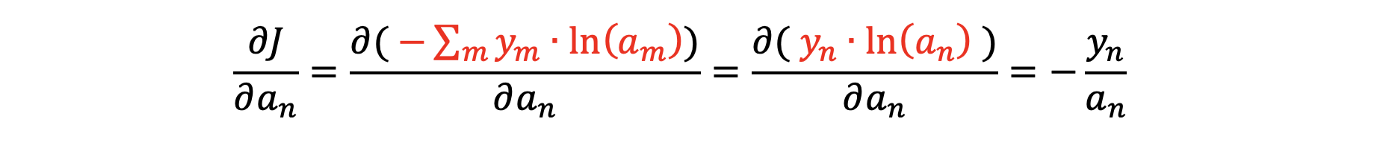
Eq 1.0

It will simplify things a bit if we write this as



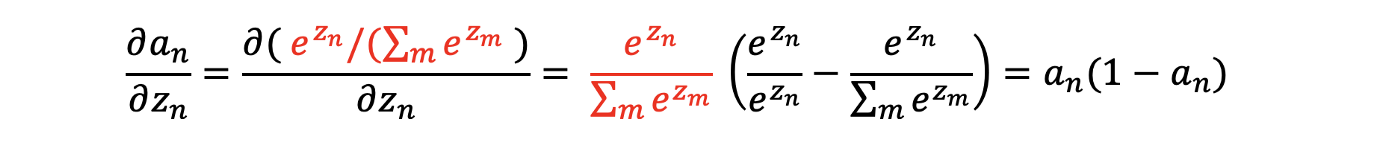
considering m=n and m≠n then adding

Starting with ∂J/∂aₙ we can write



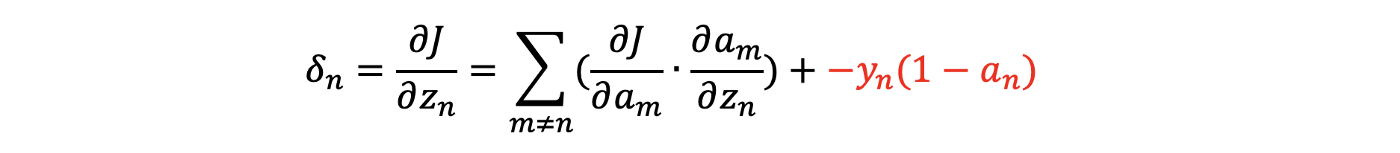
The nth neuron is the only survivor in the sum due to ∂aₙ

and for ∂aₙ/∂zₙ we have



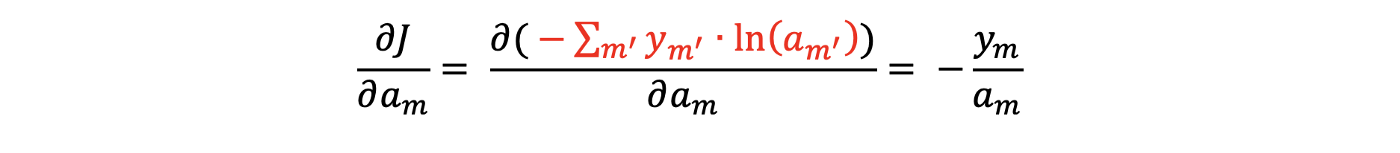
You can arrive at the derivative using the quotient rule but there are [other ways](https://essamamin99.medium.com/differentiation-revisiting-the-product-rule-112b8b0bf7b6) that might be worth checking out.

Now multiplying both of our results we and plugging in the original equation we get



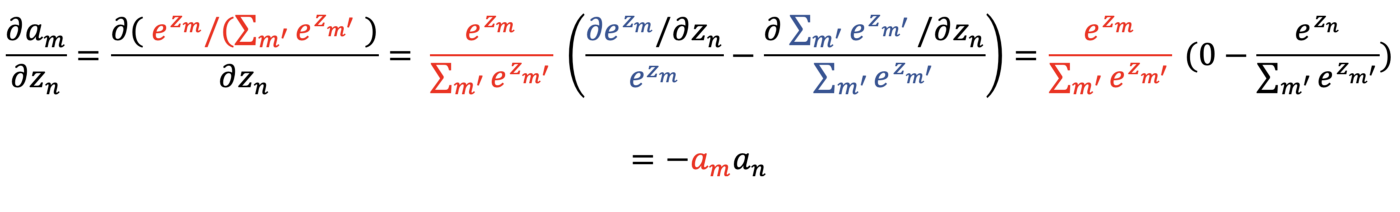
By Plugging in Eq. 1

For the remaining sum, let’s first compute ∂J/∂aₘ by writing



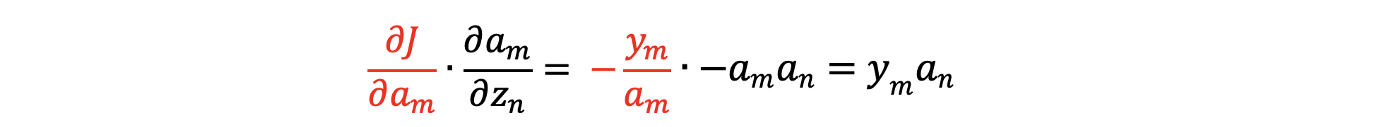
If a formula involves a sum over everything you can always change the index to avoid confusion

And then for ∂aₘ/∂zₙ (m≠n) we have

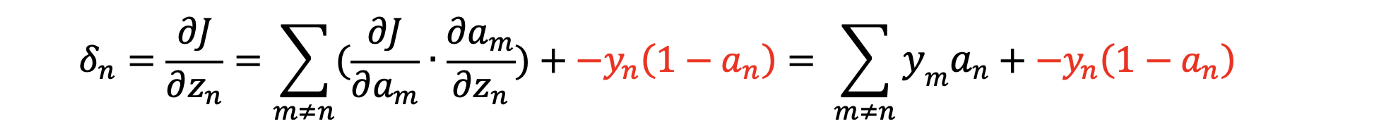


You can also use the quotient rule to arrive at the same result.

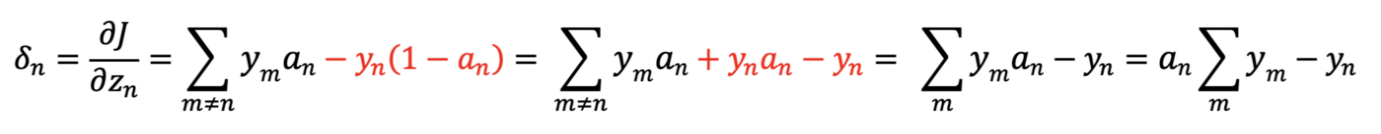
Now multiplying both results, we get



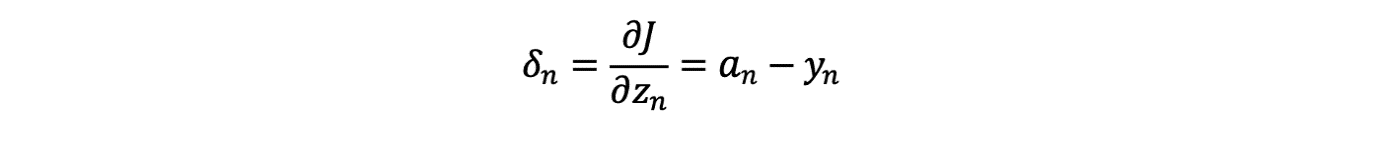
and propagating that back to the original equation we get



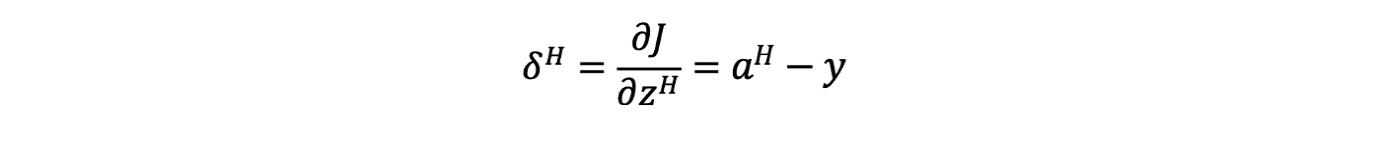
which can be simplified to



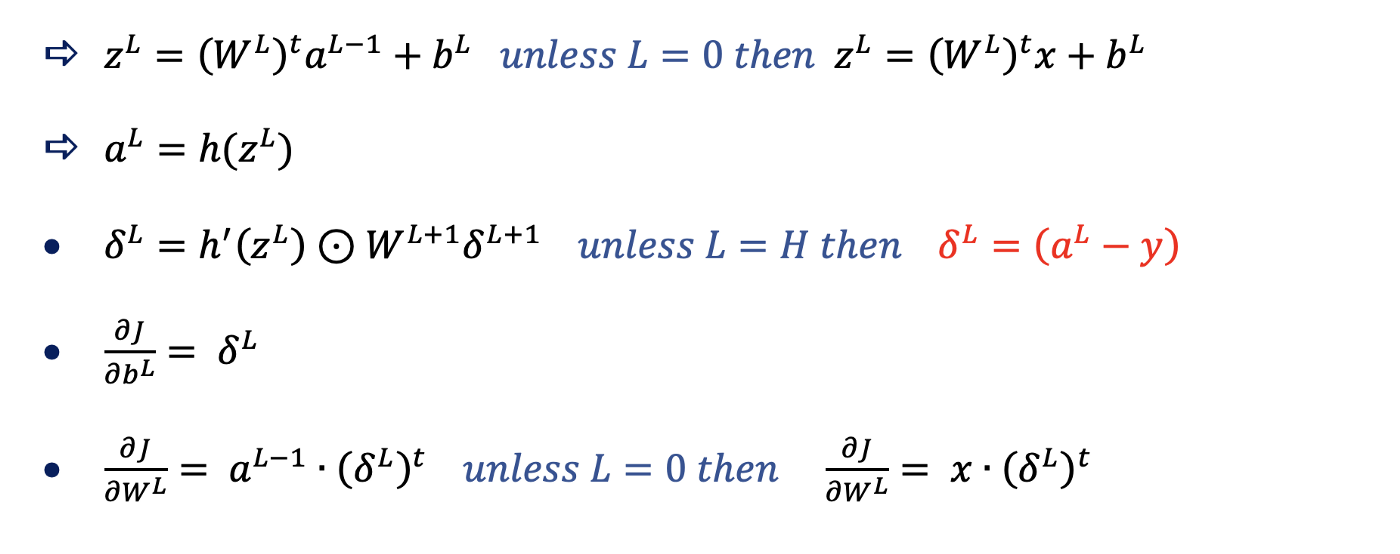
provided that y is a one-hot vector, we know that Σₘyₘ=1; thus, we can write



or in vector form after reimposing the superscript (H to denote the last layer)



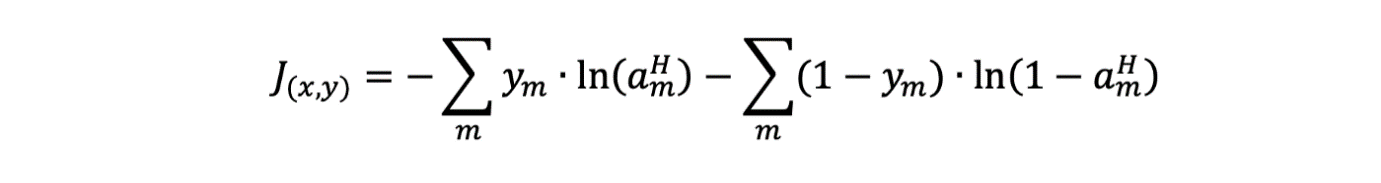
To conclude the proof, let’s update the backpropagation equations to



Only what’s in red has changed.

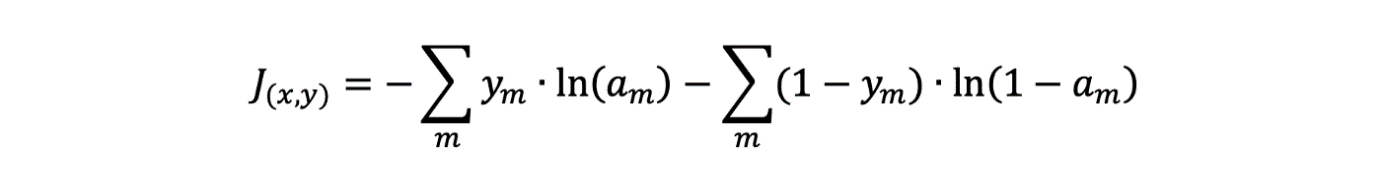
We’re not all there yet. Using categorical cross-entropy your model would do good classifying the image below as a dog; however, your dataset might include many images that have both cats and dogs in them. In this case, y is no longer a one-hot vector (and rather looks something like [0 1 0 0 1] if two classes are present within the image.) Thus, we use multi-class cross-entropy and refrain from using Softmax in the last layer; instead, we use Sigmoid. We conclude that a class is present in the image if its Sigmoid activation is greater than some threshold (0.5 for instance.)

The multi-class cross-entropy loss function for on example is given by

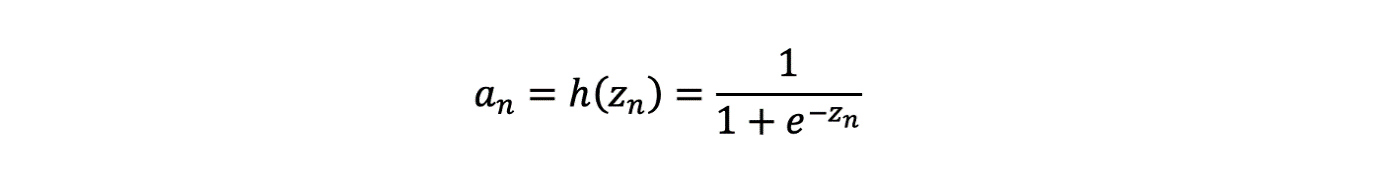


aᴴₘ is the mth neuron in the last layer (H)

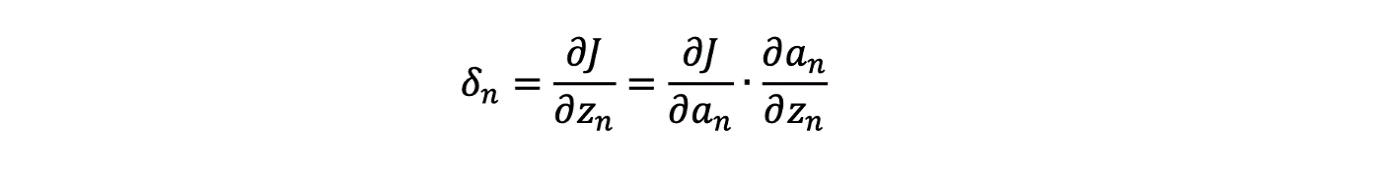
If we go back to dropping the superscript we can write



Because we’re using Sigmoid, we also have

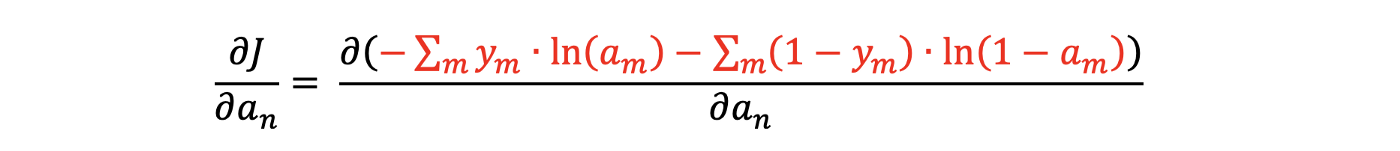


Unlike Softmax aₙ is only a function in zₙ; thus, to find δ for the last layer, all we need to consider is that

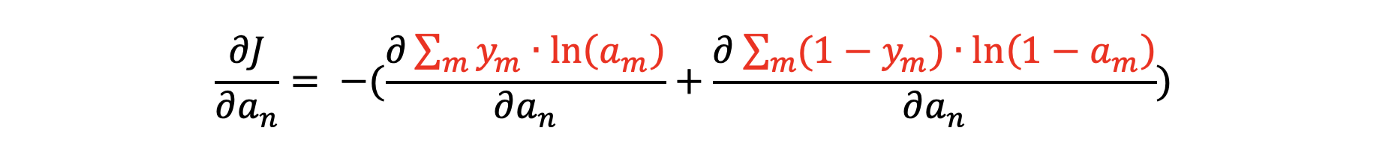


Eq. 2

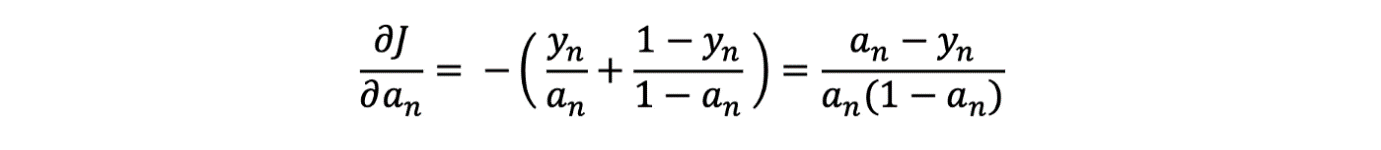
or more precisely



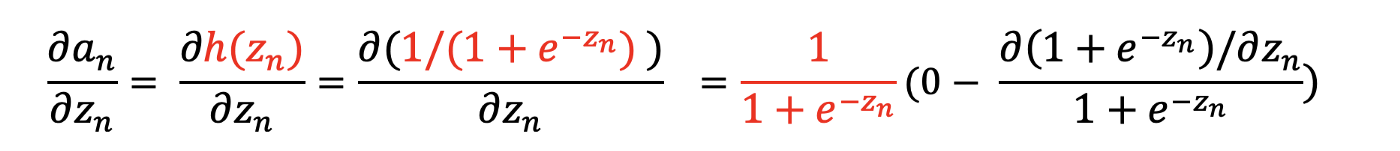
which is



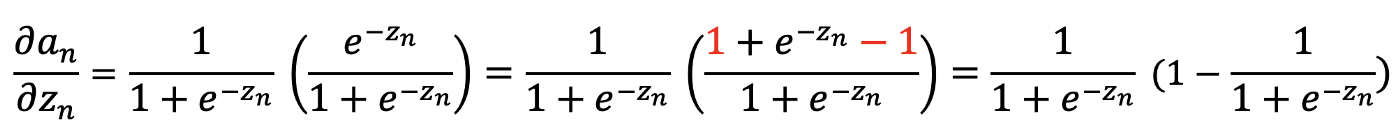
by differentiating (like we did earlier) then adding the two fractions we get



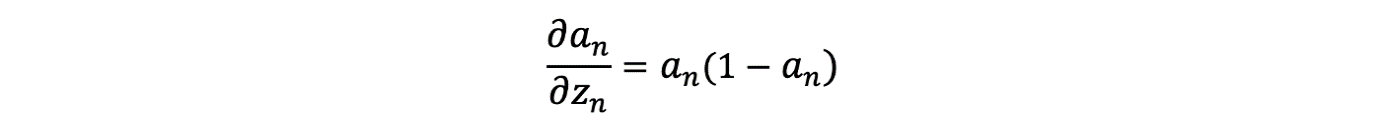
Now we need to consider ∂aₙ/∂zₙ before plugging in equation 2.0

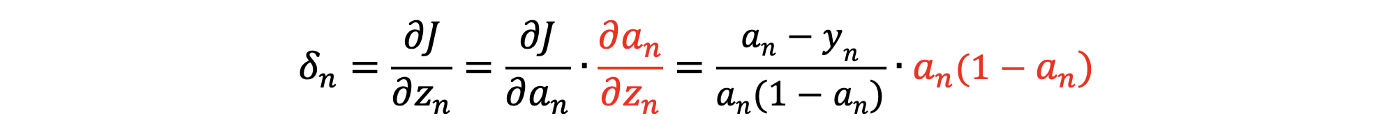


this is simply the derivative of the Sigmoid function

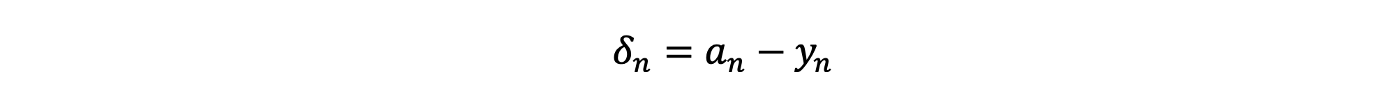


by substituting aₙ for it’s definition we get

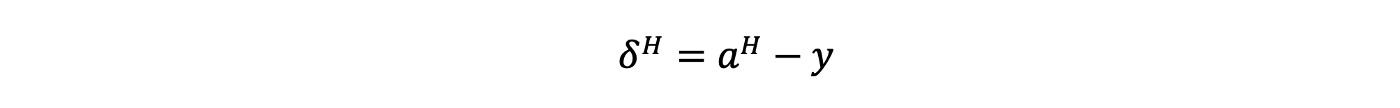
Now using both results in the original equation



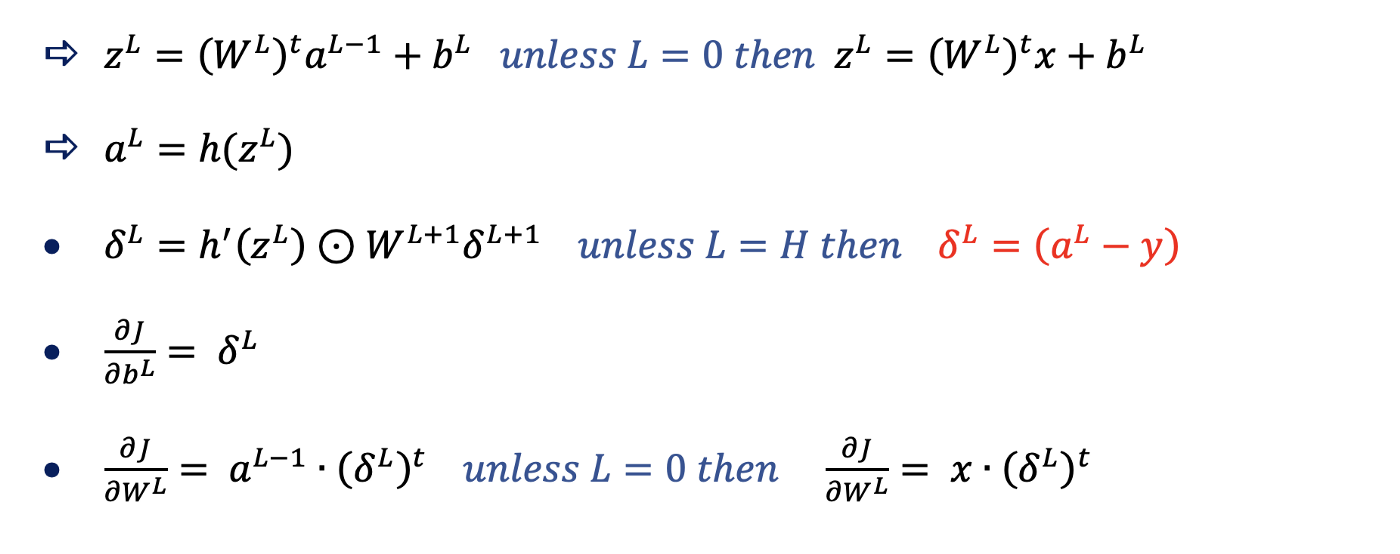
thereby,



and now by reimposing the superscript and writing this in vector form



which quite marvelously is the same result we got for categorical cross-entropy with Softmax as an activation. So we still use



as our backpropagation equations.

## References:

1. <https://towardsdatascience.com/multiclass-classification-with-support-vector-machines-svm-kernel-trick-kernel-functions-f9d5377d6f02>
2. <https://towardsdatascience.com/deriving-backpropagation-with-cross-entropy-loss-d24811edeaf9>
3. <http://neuralnetworksanddeeplearning.com/chap2.html>
4. <https://towardsdatascience.com/mse-is-cross-entropy-at-heart-maximum-likelihood-estimation-explained-181a29450a0b>
5. Youtube.com
6. NPTEL