Neural Network and deep learning Assignment

Pallavi. K. G
1JT20AI026.

1. Explain how ANN evolved from biological neurons. Illustrate how logical computations are acomplished using neurons.

Ans: ANNs were first introduced in 1943 by the neurophysiologist Warren McCulloh and the mathematician Walter Pitts. In their paper, "A logical Calculus of Ideas Immanent in Nervous Activity", McCulloh and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using propositional logic. This was the first artificial neural network architecture. Since then many architectures have been invented.

Despite early enthusiasm, interest waned until the 1980s when new architectures and better training techniques reignited attention. In the 1990's, ANNs faced competition from other methods but bounced back due to data availability, increased computing power, and improved algorithms. ANNs simulate logical computations akin to how neurons in the brain work, allowing them to tackle various tasks. Today, they're widely used for complex-problem solving despite past setbacks.

- There is now a huge quantity of data available to train neural networks and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law, but also thanks to the gaming industry, which has produced powerful GPU cards by the millions.

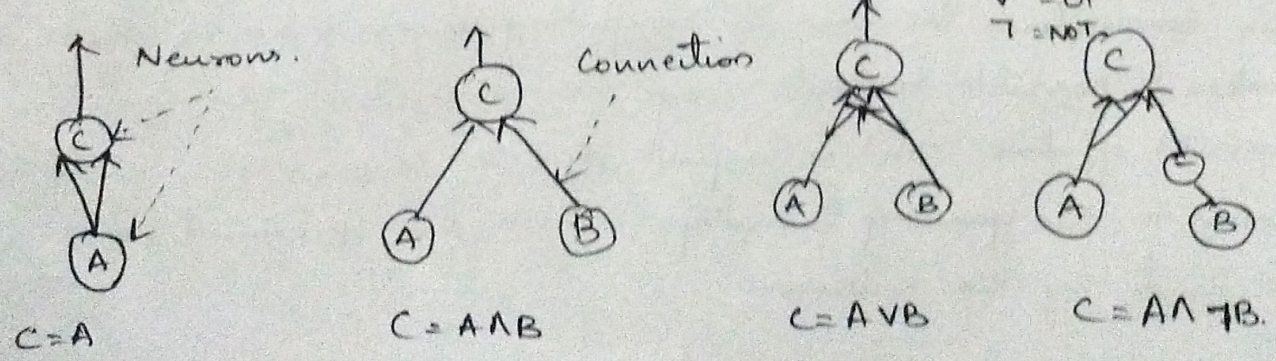
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice.
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the ~~head~~ headline news, which pulls more and more attention and funding them, resulting in more and more progress, and even more amazing products.

Logical components with Neurons:

Warren McCulloch and Walter Pitts proposed a very simple model of the biological neuron, which later became known as artificial neuron: It has one or more binary inputs and one binary output. The artificial neuron simply activates its output when more than a certain number of its inputs are active. They showed that ~~what~~ even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want.

For example, lets build a few ANN's that perform various logical computations, assuming that a neuron is activated when atleast two of its inputs are active.

$\wedge$ = AND
$\vee$ = OR
$\neg$ = NOT



Threshold logic Unit (TLU) is similar to perceptron but uses a different activation function. Instead of a step function, it uses a threshold function, where the output is 1 if the weighted sum of inputs exceeds a certain threshold, otherwise, the output is 0.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
iris = load_iris()
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.3, random_state=42)
tlu = Perceptron(activation='threshold')
tlu.fit(x_train, y_train)
accuracy = tlu.score(x_test, y_test)
print("TLU accuracy:", accuracy)
```

The perceptron learning rule is a simple algorithm used to train a perceptron. It adjusts the weights and bias of the perceptron ~~adjusts the weights as~~ iteratively to minimize the error between the predicted output and the actual output.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
iris = load_iris()
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.3, random_state=42)
class perceptron:
    def __init__(self):
        self.weights = np.random.rand(4)
        self.bias = np.random.rand()
```

- The first network on the left is simply the identity function: if neuron A is activated, then neuron C gets activated as well; but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated.
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity, then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and if neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

2. Explain perceptron, threshold logic unit, perceptron learning rule with code

Ans: A perceptron is a type of artificial neuron that takes multiple binary inputs and produces a single binary output. It calculates the weighted sum of its inputs, adds a bias, and then applies an activation function to determine the output.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                    test_size=0.2, random_state=42)
perceptron = Perceptron()
perceptron.fit(X_train, y_train)

accuracy = perceptron.score(X_test, y_test)
print("Perceptron accuracy:", accuracy)
```

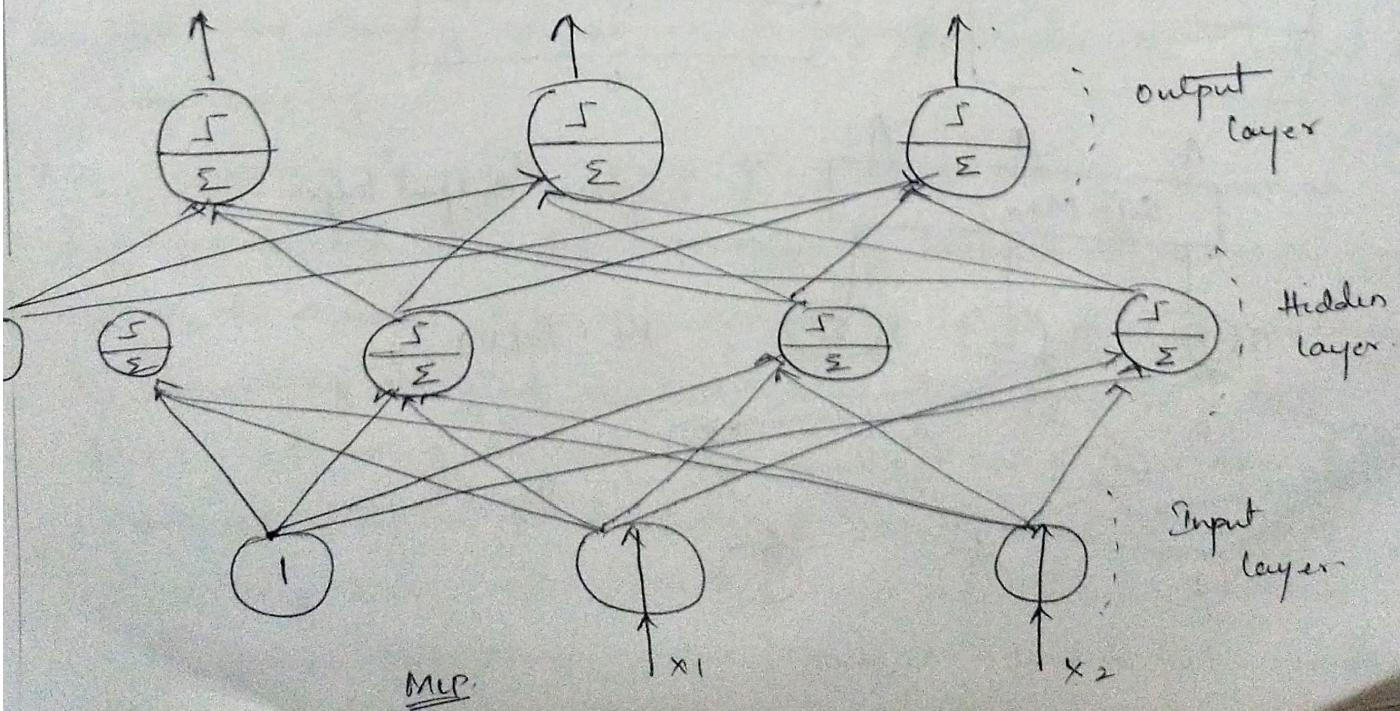
```
def predict (self, inputs):
    return int(np.dot(inputs, self.weights) + self.bias > 0)
def train(self, inputs, labels, learning_rate=0.1, epochs=100):
    for _ in range(epochs):
        for x, y in zip(inputs, labels):
            prediction = self.predict()
            error = y - prediction
            self.weights += learning_rate * error * x
            self.bias += learning_rate * error.
perceptron = Perceptron()
perceptron.train(x_train, y_train)
accuracy = np.mean([perceptron.predict(x) == y for x, y in zip(x_test,
                                                              y_test)]).
print("Perceptron with learning rule accuracy", accuracy).
```

3. Explain multi-layer perceptron and backpropagation

Ans. * An MLP consists of an i/p layer, one or more hidden layers of LTU's and an o/p layer.

* Each layer except the o/p layer, has a bias neuron and is fully connected to next layer.

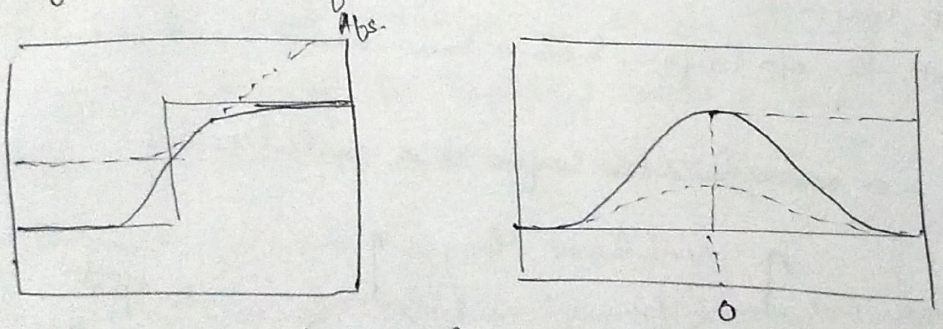
* If an MLP has 2 or more hidden layers it is called DNN.

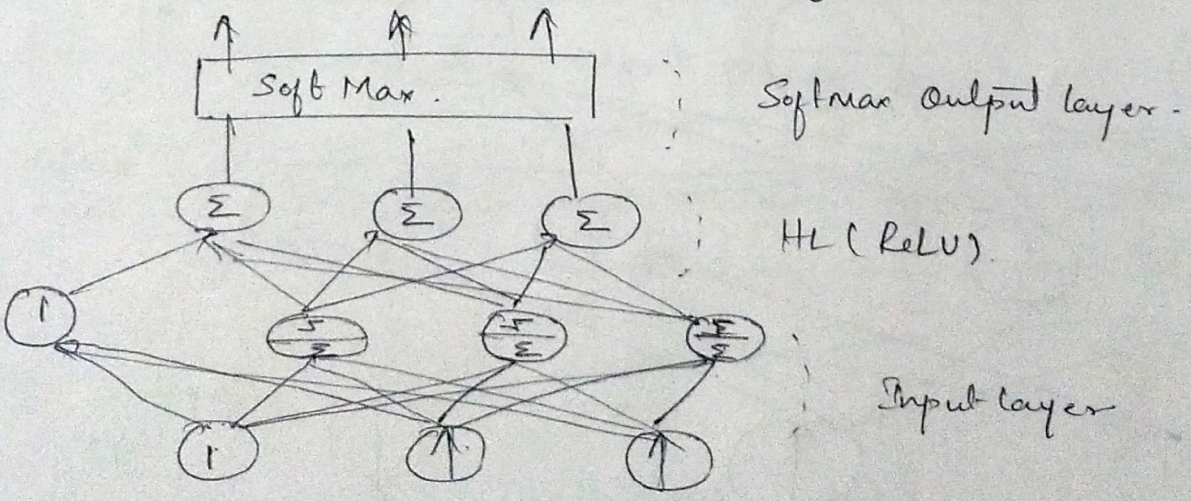


MLP.

* Backpropogation is a training algorithm for MLP's, its w[illegible] descent with reverse-mode autodiff to adjust connection weights.
* For each training instance the backpropogation algorithm first m[illegible] prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (gradient descent step).
* In order for this algorithm to work, a small change was made in the MLP's architecture: they replaced the step function with logistic function; $\sigma(z) = 1 + (1 + \exp(-z))$, because the step function contains only flat segments, so there is no gradient to work with, while the logistic function has a well defined non-zero derivate everywhere allowing gradient descent to make some progress at every step.
* The backpropogation algorithm ~~us~~ may be used with other activation functions, instead of logistic function.
* 2 popular activation functions are:
  → The hyperbolic tangent function $\tanh(z) = 2\sigma(2z) - 1$
  → The ReLU function, $ReLU(z) = \max(0, z)$

fig1: Activation function and their derivation





What are Regression MLP's, classification MLP's, illustrate how MLP's are implemented

Ans: The following code trains a DNN for classification with 2 hidden layers and a softmax output layer with 10 neurons.

```
import ~~transf~~ ~~tcon~~ tensorflow as tf
feature_columns = tf.contrib.learn.infer_real_valued_columns_
                  from_input(x_train).
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100],
          n_classes=10, feature_columns=feature_columns)
dnn_clf.fit(x=x_train, y=y_train, batch_size=50, steps=4000)
```

=> using MNIST dataset we get accuracy over 98.1%

```
from sklearn.metrics import accuracy_score.
y_pred = list(dnn_clf.predict(x_test))
accuracy_score(y_test, y_pred)
0.98180000001
```

The TF learn library also provides some convenience functions to evaluate models:

```
dnnclf.evaluate(x_test, y_test)
{'accuracy': 0.98180002, 'global_step': 4000, 'loss': 0.736}
```

6. How fine-tuning neural network hyperparameters are actuated in deep learning.

Ans: Choosing the right settings for a neural network, known as hyperparameters, can be overwhelming because there are so many options to tweak, like the number of layers or neurons per layer. It's like trying to find the best combination of settings in a video game, but using methods like random searching or specialized tools can help narrow down the choices faster. Starting with something basic, like deciding how many levels to include in the game, can simplify the process and lead to better results

in less time. lets start with the number of hidden layers

* Hidden layers: Imagine drawing a forest with just one tree. You have to draw every leaf, you can copy and paste parts make it quicker. Similarly, deep networks have layers that help understand different parts of things, like lines, shapes and faces.
* Number of neurons per layer: Its like having different sized buckets to catch rain. Start with big buckets to catch a lot, then use smaller ones as you need less.
* Activation functions: Think of ReLU like a switch that turns on when there's enough light and off when its dark. Its fast and doesn't get stuck. So, by adding more layers look at, and using the right switches we can make computers learn and understand things better

Module-2

1. Explain Glarot initialization and ~~Natur~~ Nonsaturating Activation functions.

Ans: In their paper, Glorot & Bengio propose Xavier initialization to ensure proper signal flow in both forward and backward direction in NN. They aim to maintain equal variance in the o/ps of each layer compared to its i/ps, as well as in the gradients before and after flowing through a layer during backpropagation. This initialization method also known as Glorot initialization

⇒ Normal distribution with mean 0 & SD $\sigma = \sqrt{\frac{2}{n_{i/p} + n_{o/p}}}$

⇒ a uniform distribution b/w $-r$ & $+r$, with $r = \sqrt{\frac{6}{n_{i/p} + n_{o/p}}}$

This involves randomly initializing connection weights based on the number of i/p & o/p connection of the layers. Similarly, he initialization, named after its author, is used for activation functions like ReLU to address the same variance issues. These strategies have significantly contributed to the success of deep learning by

erating training and ensuring stable signal propogation

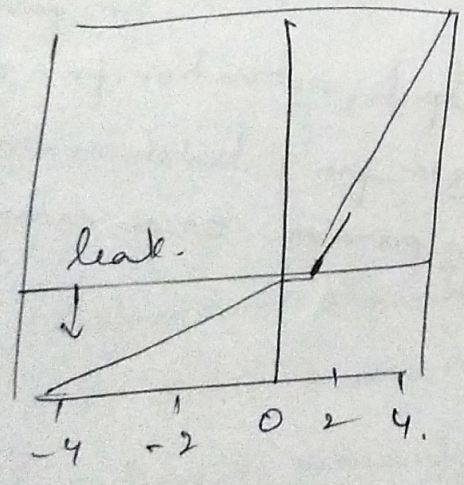
Non saturating Activation functions :

In 2010, Glorot & Benjo discovered that traditional sigmoid activation function in NNs were not ideal, leading to vanishing or exploding gradients. ReLU, a faster and non-saturating alternative, become popular.

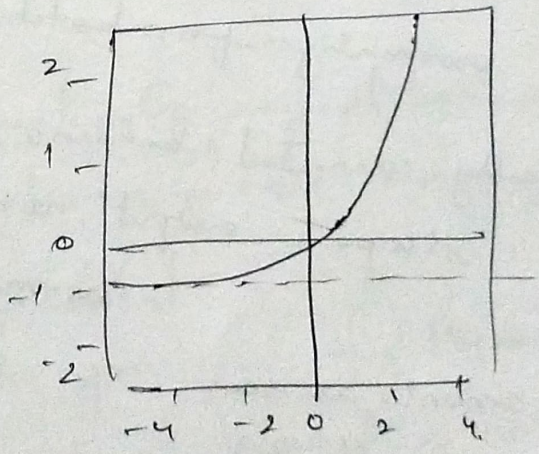
However, ReLU has its own issue called 'dying ReLU's', where some neurons stop functioning during training, outputing only zeroes when their i/p sum is negative, as the ReLU gradient is zero for -ve i/p's.

In 2015, Djork-Arne-Clevertelal introduced a novel activation function called the exponential linear unit, which surpassed all ReLU variants in their exps. The ELU not only reduced training time but also improved the NN's performance on the test set.

$$ELU_\alpha(z) = \begin{cases} \alpha(\exp(z)-1) & \text{if } z<0 \\ z & z \geq 0. \end{cases}$$



Leaky ReLU AF



ELU AF.

2. Explain batch normalization algorithm with ~~relativ~~ code.

Ans: Batch normalization is a technique used in NN's to improve training speed and stability. In tensorflow, you can implement batch normalization using the 'batch_norm()' function, which handles centering, normalizing & shifting inputs.

```
import tensorflow as tf.
from tensorflow.contrib.layers import batch_norm.
n_inputs = 28*28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10.
x = tf.placeholder(tf.float32, shape=(none, n_inputs), name='x')
is_training = tf.placeholder(tf.bool, shape(), name='is_training')
bn_params = {'is_training': is_training,
             'decay': 0.99,
             'update_collections': None
             }.

hidden1 = fully_connected(x, n_hidden1, scope='hidden1',
          normalizer_fn = batch_norm, normalizer_parms =
                          bn_params)
hidden2 = fully_connected(hidden1, n_hidden2, scope='hidden2',
          normalizer_fn = batch_from, normalizer_param =
                          bn_params)
logits = fully_connected(hidden2, n_outputs, activation_fn = none,
          scope = 'output', normalizer_fn = batch_norm)
                          normalizer_param = bn_params)

with tf.session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        for x_batch, y_batch in zip(x_batches, y_batches):
            sess.run(training_op, feed_dict = {is_training
                True, x: x_batch, y: y_batch})
        accuracy_score = accuracy.eval(feed_dict = {is_
                training: False, x: x_test_scaled,
                y: y_test})
    print(accuracy_score)
```

What are fast optimizers? Explain Momentum optimization, Nesterov Accelerated gradient, Adagrad, RMS prop and adam & nadam optimization.

Ans: Fast optimizers speed up training of deep neural network by efficiently updating the model parameters.

* Momentum Optimization:

Momentum optimization accelarates training by adding the gradient's momentum at each iteration, simulating acceleration rather than speed. It introduces a momentum hyperparameter to control the contribution of past gradients the terminal velocity or maximum weight update, is determined by the gradient, learning rate & momentum. Unlike GD, MO escapes plateau faster and ~~nag~~ navigates elongated cost function more efficiently. In tensorflow, its implemented simply by replaying 'Gradient Descent Optimizer' with 'momentum optimizer'. the momentum value of 0.9 often work well in practice.

optimizer = tf.train.MomentumOptimizer(learning_rate=
learning_rate, momentum=0.9)

* Nesterov Accelaration Gradient:

NAG measures the gradient slightly ahead in the direction of momentum, entering momentum optimization. It computes the gradient the ~~opt~~ rather than at θ. leveraging the fact that the momentum vector generally points towards the optimum.

* AdaGrad.

Eg: $s \leftarrow s + \nabla_\theta S(\theta) \otimes \nabla_\theta J(\theta)$

$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oplus \sqrt{s+\epsilon}$

Adagrad scales down the gradient vector along the steepest dimension by accumulating the square of gradient into a vector s. The gradient vector is then + element-wise by the √ of the accumulated squares of gradients, with a smoothing term to avoid division by zero.

* RMS Prop:

RMS prop accumulates only the gradients from recent iterations using exponential decay, unlike Ada grad which accumulates all gradients. The decay rate, typically set to 0.9, controls how quickly past gradients are forgotten.

1. $s \leftarrow \beta s + (1-\beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oplus \sqrt{s+\epsilon}$

* Adam Optimization:

Adam algorithm: 1. $m \leftarrow \beta m + (1-\beta) \nabla_\theta J(\theta)$

2. $s \leftarrow \beta_2 s + (1-\beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$

3. $m \leftarrow \frac{m}{1-\beta_1^T}$

4. $s \leftarrow \frac{s}{1-\beta_2^T}$

5. $\theta \leftarrow \theta - \eta m \oplus \sqrt{s+\epsilon}$

Combines the concepts of MO & RMSProp. It maintains exponentially decaying average of past gradients & squared gradients. The algorithm uses bias-corrected estimates to address initialization issues. The momentum decay hyperparameter (β_1) is typically set to 0.9 & scaling hyperparameter