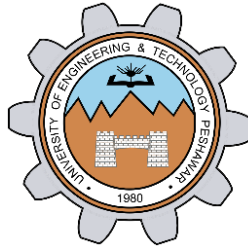


# **Introduction to Deep Learning**

## **LAB # 13**



**Fall 2021**

**Data Analytics Lab**

Submitted by: **Shah Raza**

Registration No.: **18PWCSE1658**

Class Section: **B**

“On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work.”

Student Signature: \_\_\_\_\_

Submitted to:

**Engr. Mian Ibad Ali Shah**

9 March 2022

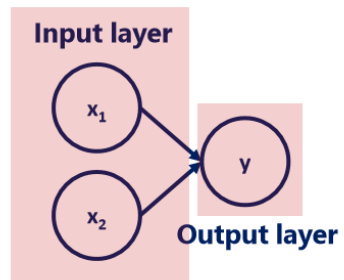
Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

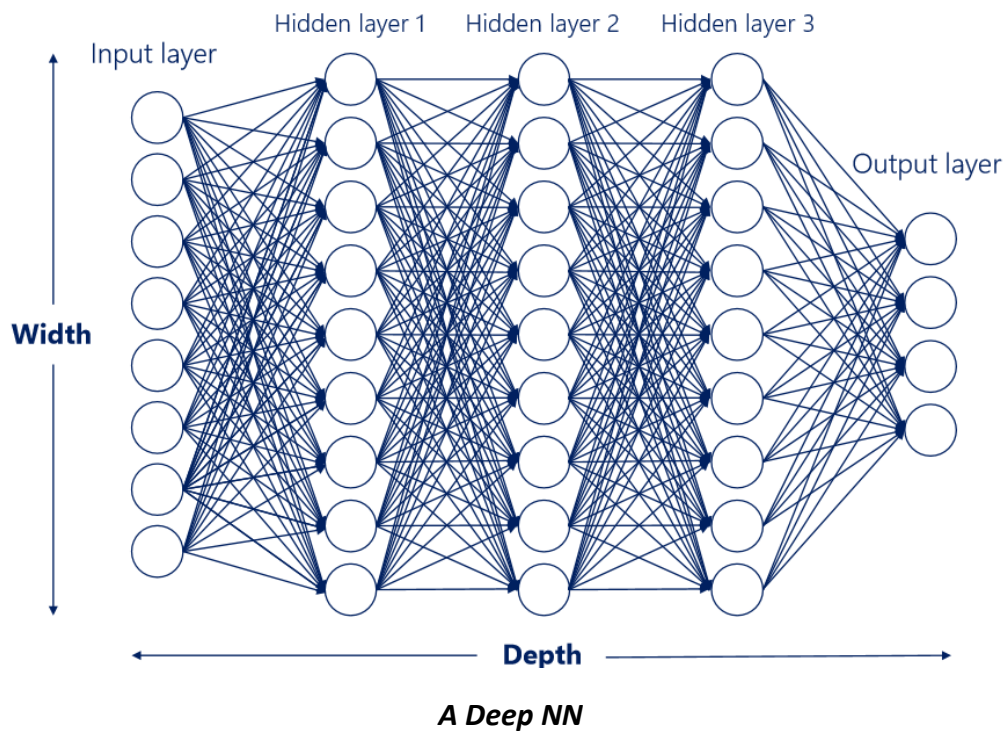
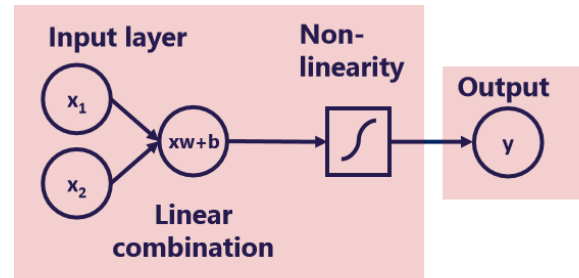
## Introduction to Deep Learning:

### Digging deeper to NN

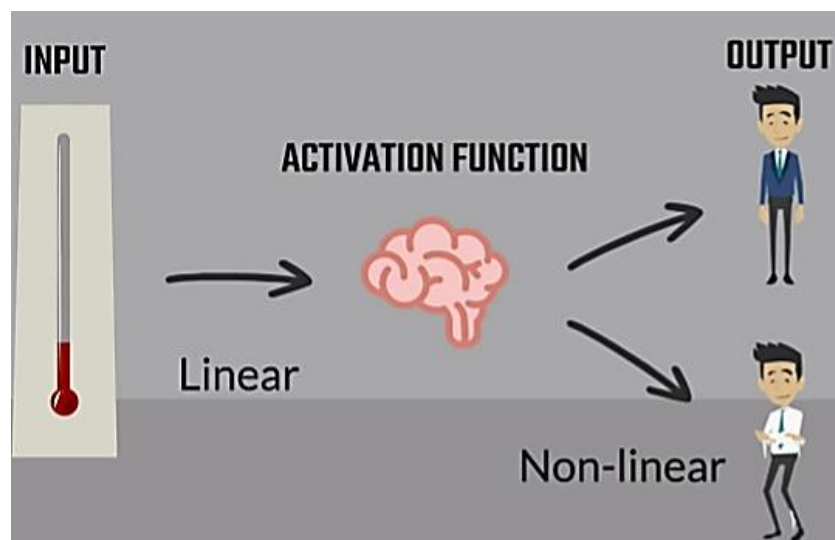
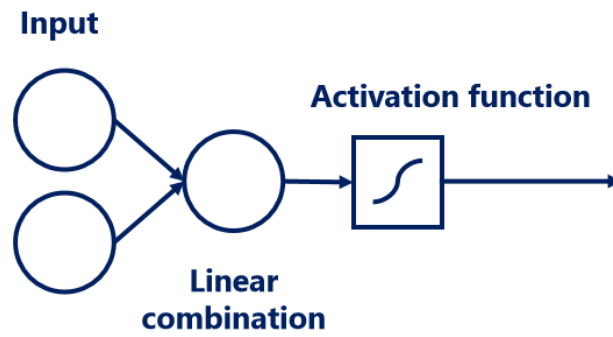
#### Minimal example (a simple neural network)



#### Neural networks

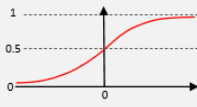
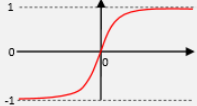
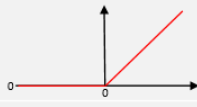


**Activation Functions:**

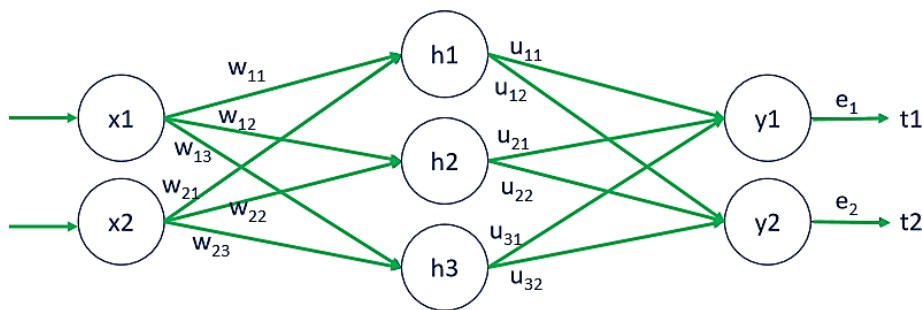


*Jacket or no jacket, based on temperature*

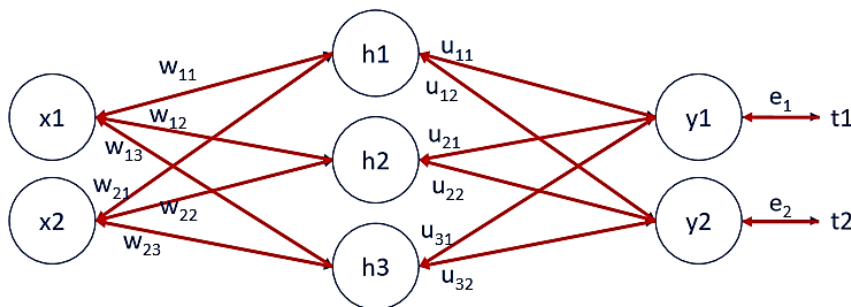
### Some activation functions:

Name	Formula	Derivative	Graph	Range
<b>sigmoid</b> (logistic function)	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$		(0,1)
<b>TanH</b> (hyperbolic tangent)	$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^a + e^{-a})^2}$		(-1,1)
<b>ReLU</b> (rectified linear unit)	$\text{relu}(a) = \max(0, a)$	$\frac{\partial \text{relu}(a)}{\partial a} = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases}$		(0,∞)
<b>softmax</b>	$\sigma_i(a) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(a)}{\partial a_j} = \sigma_i(a) (\delta_{ij} - \sigma_j(a))$ Where $\delta_{ij}$ is 1 if $i=j$ , 0 otherwise		(0,1)

### Forward & Backward Propagation:



**Forward propagation** is the process of pushing inputs through the net. At the end of each epoch, the obtained outputs are compared to targets to form the errors.

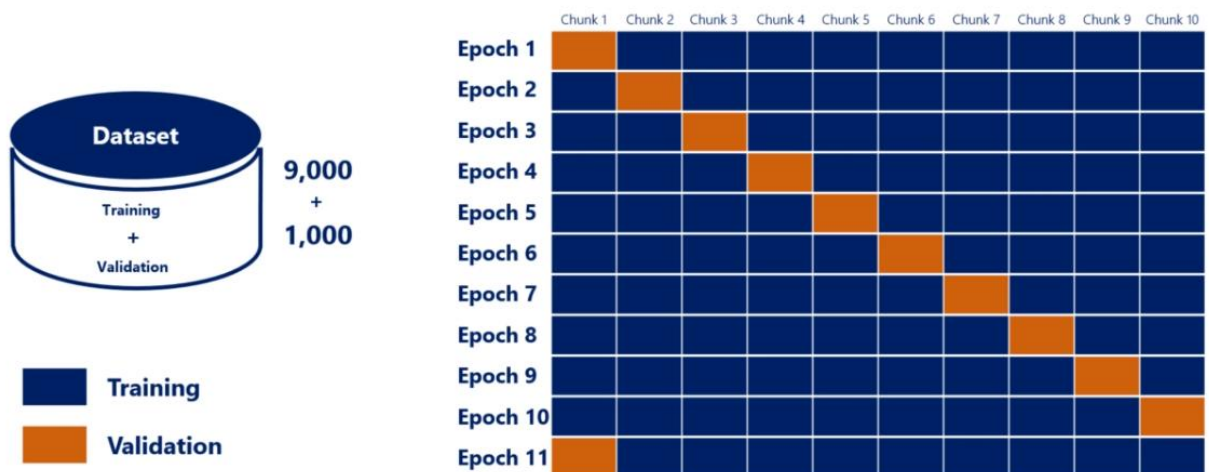


**Backpropagation** of errors is an algorithm for neural networks using gradient descent. It consists of calculating the contribution of each **parameter** to the errors. We backpropagate the **errors** through the net and **update** the parameters (weights and biases) accordingly.

## Avoiding Overfitting: Training, Validation and Test Datasets:

1. Split data into Training, Validation and Test (by a ratio of 80,10,10 or 70,20,10 etc)
2. Train the model with the training dataset only! (Backward Propagation)
3. After every epoch, validate the model using validation dataset (Only forward Propagation)
4. If the validation is increasing, stop the training! (It is going to overfitting)
5. Finally, test the model (Forward Propagation only)

N-Fold cross validation (used when we have less data) -> We combine Training and Validation datasets and use data chunks of N folds. In every epoch a new validation set (chunk of data) is selected.



## Task 1:

### Deep Neural Network for MNIST Classification

#### Import the relevant packages

```
In [1]: import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("CIFAR-10_data/", one_hot=True)
```

WARNING: Logging before flag parsing goes to stderr.  
W1231 01:47:49.868909 8732 deprecation.py:323] From <ipython-input-1-7fba8d6994c9>:4: read\_data\_sets (from tensorflow.contrib.learn.python.learn.datasets.mnist) is deprecated and will be removed in a future version.  
Instructions for updating:

```
In [9]: input_size = 784
output_size = 10
# Use same hidden layer size for both hidden layers. can be different!
hidden_layer_size = 50

# Reset any variables left in memory from previous runs.
tf.reset_default_graph()

# As in the previous lab - declare placeholders where the data will be fed into.
inputs = tf.placeholder(tf.float32, [None, input_size])
targets = tf.placeholder(tf.float32, [None, output_size])

# Weights and biases for the first linear combination between the inputs and the first hidden layer.
# Use get_variable in order to make use of the default TensorFlow initializer which is Xavier.
weights_1 = tf.get_variable("weights_1", [input_size, hidden_layer_size])
biases_1 = tf.get_variable("biases_1", [hidden_layer_size])

# Operation between the inputs and the first hidden layer.
# ReLU is chosen as an activation function. You can try playing with different non-linearities.
outputs_1 = tf.nn.relu(tf.matmul(inputs, weights_1) + biases_1)

# Weights and biases for the second linear combination.
# This is between the first and second hidden layers.
weights_2 = tf.get_variable("weights_2", [hidden_layer_size, hidden_layer_size])
biases_2 = tf.get_variable("biases_2", [hidden_layer_size])

# Operation between the first and the second hidden layers. Again, use ReLU.
outputs_2 = tf.nn.relu(tf.matmul(outputs_1, weights_2) + biases_2)

# Weights and biases for the final linear combination.
weights_3 = tf.get_variable("weights_3", [hidden_layer_size, output_size])
```

Activate Windows  
Go to Settings to activate Windows.

```
# Weights and biases for the final linear combination.
weights_3 = tf.get_variable("weights_3", [hidden_layer_size, output_size])
biases_3 = tf.get_variable("biases_3", [output_size])

outputs = tf.matmul(outputs_2, weights_3) + biases_3

# Calculate the loss function for every output/target pair.
# Logits here means: unscaled probabilities (so, the outputs, before they are scaled by the softmax)
loss = tf.nn.softmax_cross_entropy_with_logits(logits=outputs, labels=targets)

# Get the average loss
mean_loss = tf.reduce_mean(loss)

# Define the optimization step. Using adaptive optimizers such as Adam
optimizer = tf.train.AdamOptimizer(learning_rate=0.001).minimize(mean_loss)

# Get a 0 or 1 for every input in the batch indicating whether it output the correct answer out of the 10.
out_equals_target = tf.equal(tf.argmax(outputs, 1), tf.argmax(targets, 1))

# Get the average accuracy of the outputs.
accuracy = tf.reduce_mean(tf.cast(out_equals_target, tf.float32))

# Declare the session variable.
sess = tf.InteractiveSession()

# Initialize the variables. Default initializer is Xavier.
initializer = tf.global_variables_initializer()
sess.run(initializer)

# Batching
batch_size = 100
```

Activate Windows  
Go to Settings to activate Windows.

Epoch 1. Mean loss: 0.255. Validation loss: 0.175. Validation accuracy: 94.90%  
Epoch 2. Mean loss: 0.153. Validation loss: 0.140. Validation accuracy: 96.20%  
Epoch 3. Mean loss: 0.135. Validation loss: 0.156. Validation accuracy: 96.52%  
End of training.

## Test the model

```
In [6]: input_batch, target_batch = mnist.test.next_batch(mnist.test._num_examples)
test_accuracy = sess.run([accuracy],
                        feed_dict={inputs: input_batch, targets: target_batch})

test_accuracy_percent = test_accuracy[0] * 100.

print('Test accuracy: ' + '{0:.2f}'.format(test_accuracy_percent) + '%')
```

Test accuracy: 96.74%

Using the initial model and hyperparameters given in this notebook, the final test accuracy should be roughly between 97% and 98%. Each time the code is rerun, we get a different accuracy as the batches are shuffled, the weights are initialized in a different way, etc.

Finally, we have intentionally reached a suboptimal solution, so you can have space to build on it.

Activate Windows  
Go to Settings to activate Windows

