exp-1-ml

April 23, 2025

[2]:
```
import numpy as np
```

[3]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([1, 1])                    #Polar OR
theta = 1
Z = W@X
y_pred = [int(i>=theta) for i in Z]
print(y_pred)
```

```
[0, 1, 1, 1]
```

[4]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([1, 1])
theta = 2
Z = W@X                                          #Polar AND
y_pred = [int(i>=theta) for i in Z]
print(y_pred)
```

```
[0, 0, 0, 1]
```

[5]:
```
X = np.array([[-1, -1, 1, 1],
              [-1, 1, -1, 1]])
W = np.array([1, 1])
theta = 0                                       #Bipolar OR
Z = W@X
y_pred = [int(i>=theta)*2-1 for i in Z]
print(y_pred)
```

```
[-1, 1, 1, 1]
```

[6]:
```
X = np.array([[-1, -1, 1, 1],
              [-1, 1, -1, 1]])
W = np.array([1, 1])                     #Bipolar AND
theta = 1
Z = W@X
y_pred = [int(i>=theta)*2-1 for i in Z]
```

```
print(y_pred)
```

```
[-1, -1, -1, 1]
```

[ ]:

[ ]:

mlp-backward-pass-2-layered-1

April 23, 2025

[7]:
```python
import numpy as np
np.set_printoptions(precision=4)

# Initialize weights
W_0 = np.array([[1, 0, 1],
                [-1, -1, 1]], dtype=float)  # Hidden layer weights
W_1 = np.array([[0, 1, -1]], dtype=float)   # Output layer weights

# Input and target
X = np.array([[1, -1, 1],
              [0, -1, 1]], dtype=float)     # Shape: (2, 2)
t = np.array([[0, 0, 1]], dtype=float)   # Shape: (1, 2)

# Hyperparameters
f_1 = "Lin"
f_2 = "ReLU"
lr = 1
MAX_EPOCHS = 1

# Activation Functions
def USigmoid(x, direction='F'):
    fx = 1 / (1 + np.exp(-x))
    return fx if direction == 'F' else fx * (1 - fx)

def BSigmoid(x, direction='F'):
    fx = (1 - np.exp(-x)) / (1 + np.exp(-x))
    return fx if direction == 'F' else 0.5 * (1 - fx ** 2)

def TanH(x, direction='F'):
    fx = np.tanh(x)
    return fx if direction == 'F' else 1 - fx ** 2

def ReLU(x, direction='F'):
    return np.maximum(0, x) if direction == 'F' else (x > 0).astype(float)

def Lin(x, direction='F'):
    return x if direction == 'F' else np.ones_like(x)
```

```
# Wrapper for activation
def activation(Z, fcn="Lin", direction='F'):
    return np.array([globals()[fcn](z, direction) for z in Z]).reshape(-1, 1)

A_0 = np.vstack((np.ones((1, 1)), x.reshape(-1, 1)))

# Training loop
for ep in range(MAX_EPOCHS):
    print('\nEPOCH-', ep + 1, '=' * 80)
    for itr, (x, y) in enumerate(zip(X.T, t.T)):
        print(f'\nITER-{itr + 1} ' + '-' * 80)

        # Forward Pass: Input -> Hidden
        A_0 = x.reshape(-1, 1)

        # Conditionally add bias to A_0
        if W_0.shape[1] == A_0.shape[0] + 1:
            A_0 = np.vstack((np.ones((1, 1)), A_0))  # Add bias at the top
            print('Bias added to A_0')

        Z_1 = W_0 @ A_0
        A_1 = activation(Z_1, f_1)

        # Forward Pass: Hidden -> Output
        # Conditionally add bias to A_1
        if W_1.shape[1] == A_1.shape[0] + 1:
            A_1 = np.vstack((np.ones((1, 1)), A_1))  # Add bias at the top
            print('Bias added to A_1')

        Z_2 = W_1 @ A_1
        A_2 = activation(Z_2, f_2)

        print(f'A_0 (input):\n{A_0}')
        print(f'Z_1 (hidden pre-activation):\n{Z_1}')
        print(f'A_1 (hidden post-activation):\n{A_1}')
        print(f'Z_2 (output pre-activation):\n{Z_2}')
        print(f'A_2 (output post-activation):\n{A_2}')

       # Backward Pass: Output -> Hidden
        delta_2 = (A_2 - y.reshape(-1, 1)) * activation(Z_2, f_2, 'B')
        dW_1 = delta_2 @ A_1.T

        # Remove bias from W_1 if necessary
        if W_1.shape[1] == Z_1.shape[0] + 1:
            W_1_no_bias = W_1[:, 1:]
            print("Bias column removed from W_1 for backpropagation.")
```

```
        else:
            W_1_no_bias = W_1

        print(W_1_no_bias)

        # Backward Pass: Hidden -> Input
        delta_1 = (W_1_no_bias.T @ delta_2) * activation(Z_1, f_1, 'B')
        dW_0 = delta_1 @ A_0.T

        # Update weights
        W_1 -= lr * dW_1
        W_0 -= lr * dW_0

        print(f'dW_1:\n{dW_1}')
        print(f'dW_0:\n{dW_0}')
        print(f'Updated W_1:\n{W_1}')
        print(f'Updated W_0:\n{W_0}')
```

```
EPOCH- 1
================================================================================

ITER-1
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[1.]
 [1.]
 [0.]]
Z_1 (hidden pre-activation):
[[ 1.]
 [-2.]]
A_1 (hidden post-activation):
[[ 1.]
 [ 1.]
 [-2.]]
Z_2 (output pre-activation):
[[3.]]
A_2 (output post-activation):
[[3.]]
Bias column removed from W_1 for backpropagation.
[[ 1. -1.]]
dW_1:
[[ 3.  3. -6.]]
dW_0:
[[ 3.  3.  0.]
```

```
 [-3. -3.  0.]]
Updated W_1:
[[-3. -2.  5.]]
Updated W_0:
[[-2. -3.  1.]
 [ 2.  2.  1.]]

ITER-2
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[ 1.]
 [-1.]
 [-1.]]
Z_1 (hidden pre-activation):
[[ 0.]
 [-1.]]
A_1 (hidden post-activation):
[[ 1.]
 [ 0.]
 [-1.]]
Z_2 (output pre-activation):
[[-8.]]
A_2 (output post-activation):
[[0.]]
Bias column removed from W_1 for backpropagation.
[[-2.  5.]]
dW_1:
[[0. 0. 0.]]
dW_0:
[[0. 0. 0.]
 [0. 0. 0.]]
Updated W_1:
[[-3. -2.  5.]]
Updated W_0:
[[-2. -3.  1.]
 [ 2.  2.  1.]]

ITER-3
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[1.]
 [1.]
 [1.]]
Z_1 (hidden pre-activation):
```

```
[[-4.]
 [ 5.]]
A_1 (hidden post-activation):
[[ 1.]
 [-4.]
 [ 5.]]
Z_2 (output pre-activation):
[[30.]]
A_2 (output post-activation):
[[30.]]
Bias column removed from W_1 for backpropagation.
[[-2.  5.]]
dW_1:
[[  29. -116.  145.]]
dW_0:
[[-58. -58. -58.]
 [145. 145. 145.]]
Updated W_1:
[[ -32.  114. -140.]]
Updated W_0:
[[  56.   55.   59.]
 [-143. -143. -144.]]
```

gradient-tape-activations7

April 23, 2025

[ ]:
```
# import numpy as np

# # Initial weights
# W_0 = np.array([[1, 0, 1]], dtype=float)
# print("Initial W_0 shape:", W_0.shape)  # (1, 3)

# # Targets
# t = np.array([[1 ,-1 ,-1]], dtype=float)

# # Inputs
# X = np.array([
#     [1, 1, -1],
#     [0, 1, 1]
# ], dtype=float)
# print("X shape:", X.shape)  # (2, 3)

# # Activation function name
# f_1 = "Lin"

# # Learning rate
# lr = 0.1

# # Activation Functions
# def USigmoid(x, direction):
#     if direction == 'F':
#         return 1 / (1 + np.exp(-x))
#     else:  # derivative
#         fx = USigmoid(x, 'F')
#         return fx * (1 - fx)

# def BSigmoid(x, direction):
#     if direction == 'F':
#         return (1 - np.exp(-x)) / (1 + np.exp(-x))
#     else:
#         fx = BSigmoid(x, 'F')
#         return 0.5 * (1 - fx ** 2)
```

```
# def ReLU(x, direction):
#     if direction == 'F':
#         return np.maximum(0, x)
#     else:
#         return float(x > 0)

# def Lin(x, direction):
#     if direction == 'F':
#         return x
#     else:
#         return 1

# # Activation wrapper
# def activation(Z, fcn="Lin", direction='F'):
#     Z = np.atleast_1d(Z)
#     return np.array([globals()[fcn](z, direction) for z in Z])

# # Training loop
# MAX_EPOCH = 1

# # Pad bias term to input
# print('Pad bias at top of input')
# A_0 = np.vstack((np.ones((1, X.shape[1])), X))
# print(A_0)

# for ep in range(MAX_EPOCH):
#     print('\nEPOCH-', ep + 1, '=' * 80)
#     for itr, (x, y) in enumerate(zip(A_0.T, t.T)):
#         print('\nITER-', itr + 1, '-' * 80)

#         print(f'{y = }')

#         # Forward pass
#         print('Input -> Output')
#         Z_1 = W_0 @ x
#         print(f'{Z_1 = }')
#         A_1 = activation(Z_1, f_1)
#         print(f'{A_1 = }')

#         # Backward pass
#         print('\nOutput -> Input')
#         Error = 0.5 * (A_1 - y) ** 2
#         print(f'{Error = }')
#         dE_dW = (A_1 - y) * activation(Z_1, f_1, 'B') * x
#         print(f'{dE_dW = }')

#         # Weight update
```

```
#         W_0 = W_0 - lr * dE_dW
#         print(f'{W_0 = }')
```

```
Initial W_0 shape: (1, 3)
X shape: (2, 3)
Pad bias at top of input
[[ 1.  1.  1.]
 [ 1.  1. -1.]
 [ 0.  1.  1.]]

EPOCH- 1
================================================================================

ITER- 1
--------------------------------------------------------------------------------
y = array([1.])
Input -> Output
Z_1 = array([1.])
A_1 = array([1.])

Output -> Input
Error = array([0.])
dE_dW = array([0., 0., 0.])
W_0 = array([[1., 0., 1.]])

ITER- 2
--------------------------------------------------------------------------------
y = array([-1.])
Input -> Output
Z_1 = array([2.])
A_1 = array([2.])

Output -> Input
Error = array([4.5])
dE_dW = array([3., 3., 3.])
W_0 = array([[ 0.7, -0.3,  0.7]])

ITER- 3
--------------------------------------------------------------------------------
y = array([-1.])
Input -> Output
Z_1 = array([1.7])
A_1 = array([1.7])

Output -> Input
Error = array([3.645])
dE_dW = array([ 2.7, -2.7,  2.7])
```

```
W_0 = array([[ 0.43, -0.03,  0.43]])
```

[4]:
```python
import numpy as np
import tensorflow as tf

# Initial weights
W_0 = tf.Variable([[1.0, 0.0, 1.0]], dtype=tf.float32)
print("Initial W_0 shape:", W_0.shape)

# Targets
t = tf.constant([[1.0, -1.0, -1.0]], dtype=tf.float32)

# Inputs
X = tf.constant([
    [1.0, 1.0, -1.0],
    [0.0, 1.0, 1.0]
], dtype=tf.float32)
print("X shape:", X.shape)

# Activation function name
f_1 = "Lin"

# Learning rate
lr = 0.1

# Activation Functions
def USigmoid(x):
    return tf.math.sigmoid(x)

def USigmoid_deriv(x):
    fx = tf.math.sigmoid(x)
    return fx * (1 - fx)

def BSigmoid(x):
    return (1 - tf.exp(-x)) / (1 + tf.exp(-x))

def BSigmoid_deriv(x):
    fx = BSigmoid(x)
    return 0.5 * (1 - tf.square(fx))

def ReLU(x):
    return tf.nn.relu(x)

def ReLU_deriv(x):
    return tf.cast(x > 0, tf.float32)

def Lin(x):
```

```
    return x

def Lin_deriv(x):
    return tf.ones_like(x)

# Activation function dictionary
activation_map = {
    "USigmoid": (USigmoid, USigmoid_deriv),
    "BSigmoid": (BSigmoid, BSigmoid_deriv),
    "ReLU": (ReLU, ReLU_deriv),
    "Lin": (Lin, Lin_deriv),
}

# Select activation function and its derivative
activation_fn, activation_deriv = activation_map[f_1]

# Add bias term to input (pad bias row at the top)
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
A_0 = tf.constant(A_0, dtype=tf.float32)

# Training loop
MAX_EPOCH = 1

# Epoch loop
# Epoch loop
for ep in range(MAX_EPOCH):
    print(f"\nEPOCH-{ep + 1} " + "=" * 80)

    for itr, (x, y) in enumerate(zip(tf.transpose(A_0), tf.transpose(t))):
        print(f"\nITER-{itr + 1} " + "-" * 80)

        # Forward pass
        Z_1 = tf.matmul(W_0, tf.reshape(x, [-1, 1]))
        A_1 = activation_fn(Z_1)

        print(f"Target (y): {y.numpy()[0]:.2f}")
        print(f"Z_1 (Weighted sum): {Z_1.numpy()[0][0]:.4f}")
        print(f"A_1 (Activated output): {A_1.numpy()[0][0]:.4f}")

        # Error and gradient
        Error = 0.5 * (A_1 - y) ** 2
        dE_dZ = (A_1 - y) * activation_deriv(Z_1)
        dE_dW = tf.matmul(dE_dZ, tf.reshape(x, [1, -1]))

        print(f"Error: {Error.numpy()[0][0]:.4f}")
        print(f"dE_dW (Gradients): {np.round(dE_dW.numpy(), 4)}")
```

```
        # Update weights
        W_0.assign_sub(lr * dE_dW)
        print(f"Updated W_0: {np.round(W_0.numpy(), 4)}")
```

```
Initial W_0 shape: (1, 3)
X shape: (2, 3)

EPOCH-1
================================================================================

ITER-1
--------------------------------------------------------------------------------
Target (y): 1.00
Z_1 (Weighted sum): 1.0000
A_1 (Activated output): 1.0000
Error: 0.0000
dE_dW (Gradients): [[0. 0. 0.]]
Updated W_0: [[1. 0. 1.]]

ITER-2
--------------------------------------------------------------------------------
Target (y): -1.00
Z_1 (Weighted sum): 2.0000
A_1 (Activated output): 2.0000
Error: 4.5000
dE_dW (Gradients): [[3. 3. 3.]]
Updated W_0: [[ 0.7 -0.3  0.7]]

ITER-3
--------------------------------------------------------------------------------
Target (y): -1.00
Z_1 (Weighted sum): 1.7000
A_1 (Activated output): 1.7000
Error: 3.6450
dE_dW (Gradients): [[ 2.7 -2.7  2.7]]
Updated W_0: [[ 0.43 -0.03  0.43]]
```

22CE1261 of cat vs dog

File Edit View Insert Runtime Tools Help

Share

Commands + Code + Text

Files

Analyze your files with code written by Gemini Upload

dogs_vs_cats
sample_data
test
train
cat.jpg
dog.jpg
dogs-vs-cats.zip
kaggle.json

```
[1] !mkdir -p ~/.kaggle
    !cp kaggle.json ~/.kaggle/

[2] !kaggle datasets download -d salader/dogs-vs-cats

Warning: Your Kaggle API key is readable by other users on this system! To fix this, you can run 'chmod 600 /root/.kaggle/kaggle.json'
Dataset URL: https://www.kaggle.com/datasets/salader/dogs-vs-cats
License(s): unknown

[3] import zipfile
    zip_ref = zipfile.ZipFile('/content/dogs-vs-cats.zip', 'r')
    zip_ref.extractall('/content')
    zip_ref.close()

[4] import tensorflow as tf
    from tensorflow import keras
    from keras import Sequential
    from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, BatchNormalization, Dropout, Input, SeparableConv2D, GlobalAveragePooling2D # Added Input, SeparableConv2D, and Glo

[5] #generators
    train_ds = keras.utils.image_dataset_from_directory(
        directory = '/content/train',
```

22CE1261 of cat vs dog

File Edit View Insert Runtime Tools Help

Share Gemini

Commands + Code + Text

T4 RAM Disk

```
[5] #generators
    train_ds = keras.utils.image_dataset_from_directory(
        directory = '/content/train',
        labels='inferred',
        label_mode = 'int',  # Change to 'categorical'
        batch_size=32,
        image_size=(256,256))

    validation_ds = keras.utils.image_dataset_from_directory(
        directory = '/content/test',
        labels='inferred',
        label_mode = 'int',  # Change to 'categorical'
        batch_size=32,
        image_size=(256,256))
```

```
Found 20000 files belonging to 2 classes.
Found 5000 files belonging to 2 classes.
```

```
[6] def process(image,label):
        image = tf.cast(image/255. ,tf.float32)
        return image,label

    train_ds = train_ds.map(process)
    validation_ds = validation_ds.map(process)
```

22CE1261 of cat vs dog

File Edit View Insert Runtime Tools Help

Share Gemini

Commands + Code + Text

T4 RAM Disk

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Dense, Dropout, BatchNormalization

model = Sequential()

# Conv Block 1
model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same', input_shape=(256, 256, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 2
model.add(Conv2D(72, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 3
model.add(Conv2D(144, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 4
model.add(Conv2D(220, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 5 (newly added for depth)
model.add(Conv2D(280, kernel_size=(3,3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Global Pooling instead of Flatten
model.add(GlobalAveragePooling2D())

# Bigger Dense layer + more Dropout for regularization
model.add(Dense(96, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

Connected to Python 3 Google Compute Engine backend (GPU)

22CE1261 of cat vs dog

File Edit View Insert Runtime Tools Help

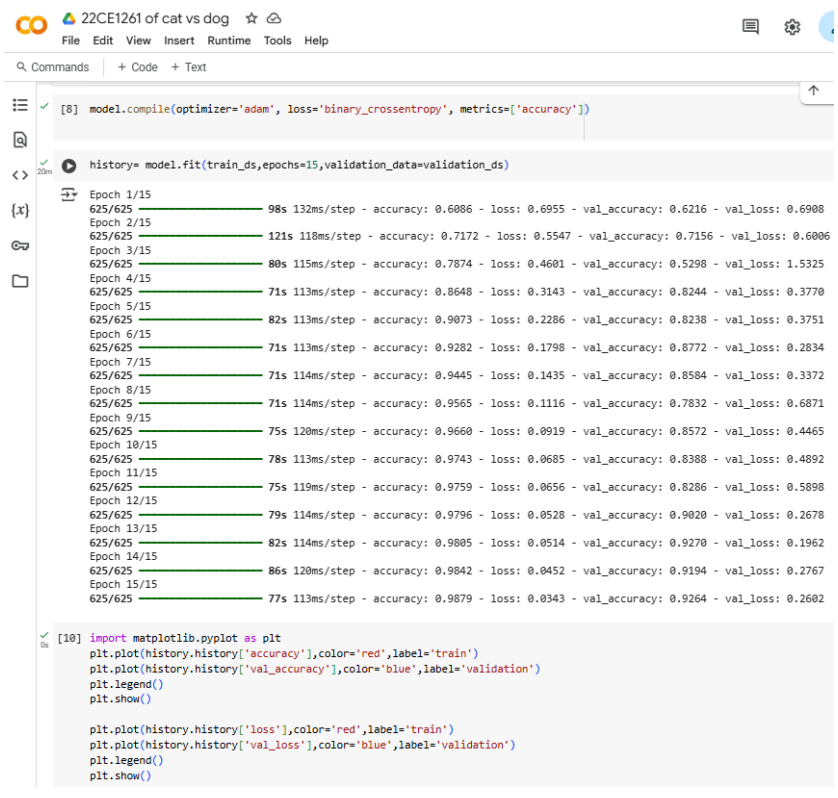
Commands + Code + Text

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 40)	1,120
batch_normalization (BatchNormalization)	(None, 256, 256, 40)	160
max_pooling2d (MaxPooling2D)	(None, 128, 128, 40)	0
conv2d_1 (Conv2D)	(None, 128, 128, 72)	25,992
batch_normalization_1 (BatchNormalization)	(None, 128, 128, 72)	288
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 72)	0
conv2d_2 (Conv2D)	(None, 64, 64, 144)	93,456
batch_normalization_2 (BatchNormalization)	(None, 64, 64, 144)	576
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 144)	0
conv2d_3 (Conv2D)	(None, 32, 32, 220)	285,340
batch_normalization_3 (BatchNormalization)	(None, 32, 32, 220)	880
max_pooling2d_3 (MaxPooling2D)	(None, 16, 16, 220)	0
conv2d_4 (Conv2D)	(None, 16, 16, 280)	554,680
batch_normalization_4 (BatchNormalization)	(None, 16, 16, 280)	1,120
max_pooling2d_4 (MaxPooling2D)	(None, 8, 8, 280)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 280)	0
dense (Dense)	(None, 96)	26,976
dropout (Dropout)	(None, 96)	0
dense_1 (Dense)	(None, 1)	97

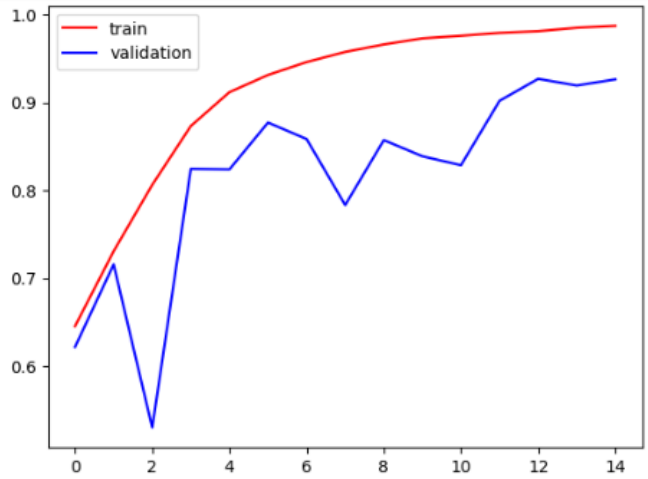
Total params: 990,685 (3.78 MB)
Trainable params: 989,173 (3.77 MB)
Non-trainable params: 1,512 (5.91 KB)

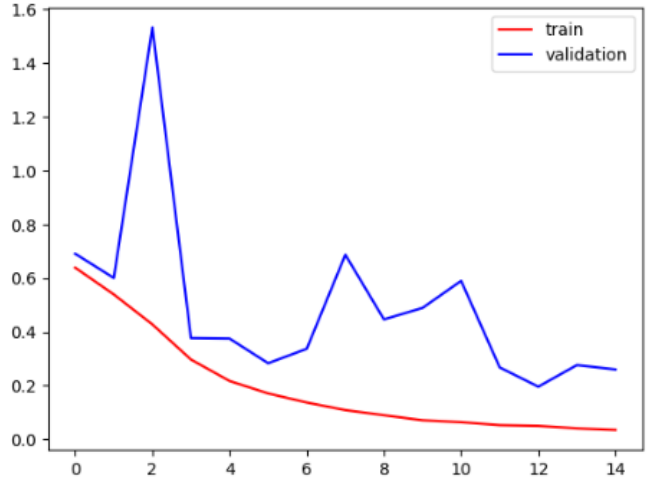


22CE1261 of cat vs dog

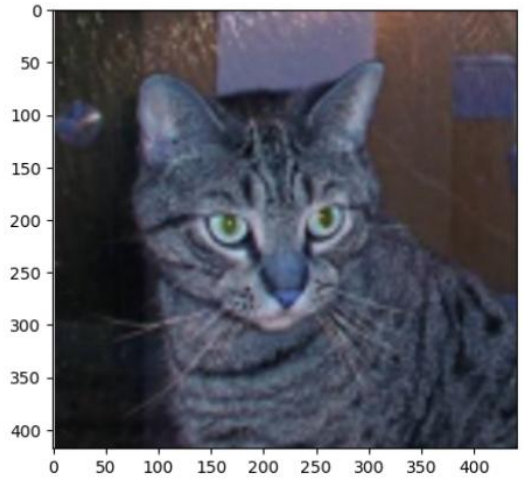
File Edit View Insert Runtime Tools Help

Commands + Code + Text



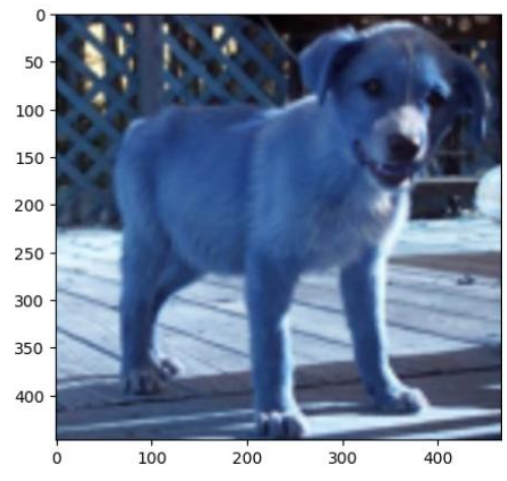



```
import cv2
test_img=cv2.imread('/content/cat.jpg')
plt.imshow(test_img)
plt.show()
test_img.shape
test_img=cv2.resize(test_img,(256,256))
test_input=test_img.reshape((1,256,256,3))
model.predict(test_input)
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
array([[0.]], dtype=float32)
```

```
import cv2
test_img=cv2.imread('/content/dog.jpg')
plt.imshow(test_img)
plt.show()
test_img.shape
test_img=cv2.resize(test_img,(256,256))
test_input=test_img.reshape((1,256,256,3))
model.predict(test_input)
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 39ms/step
array([[1.]], dtype=float32)
```

perceptron

April 22, 2025

[9]:
```python
import numpy as np

# Input data and labels
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input features
y = np.array([0, 1, 1, 1])  # Labels

# Initialize parameters
w = np.random.rand(2)  # Random initial weights
b = np.random.rand()   # Random initial bias
learning_rate = 1.0    # Learning rate
max_epochs = 1000      # Maximum number of epochs
converged = False      # Convergence flag

# Perceptron learning algorithm
for epoch in range(max_epochs):
    errors = 0
    for i in range(len(X)):
        # Compute the prediction using the dot product
        prediction = np.dot(X[i], w) + b
        # Check if the prediction matches the label
        if (y[i] == 1 and prediction <= 0) or (y[i] == 0 and prediction > 0):
            # Update weights and bias
            if y[i] == 1:
                w += learning_rate * X[i]
                b += learning_rate
            else:
                w -= learning_rate * X[i]
                b -= learning_rate
            errors += 1
    # Check for convergence
    if errors == 0:
        converged = True
        print(f"Converged after {epoch+1} epochs")
        break

if not converged:
    print(f"Did not converge after {max_epochs} epochs")
```

```
# Output the learned weights and bias
print("Learned weights:", w)
print("Learned bias:", b)

# Testing the perceptron
print("Testing:")
for i in range(len(X)):
    prediction = np.dot(X[i], w) + b
    print(f"Input: {X[i]}, Prediction: {1 if prediction > 0 else 0}, Actual:␣
 ↪{y[i]}")
```

```
Converged after 3 epochs
Learned weights: [0.93321385 1.37331598]
Learned bias: -0.5922852074362165
Testing:
Input: [0 0], Prediction: 0, Actual: 0
Input: [0 1], Prediction: 1, Actual: 1
Input: [1 0], Prediction: 1, Actual: 1
Input: [1 1], Prediction: 1, Actual: 1
```

[ ]:

```
unicodedata2                  15.1.0
Unidecode                     1.2.0
urllib3                       2.2.2
w3lib                         2.1.2
watchdog                      4.0.1
wcwidth                       0.2.5
webencodings                  0.5.1
websocket-client              1.8.0
Werkzeug                      3.0.3
whatthepatch                  1.0.2
wheel                         0.43.0
widgetsnbextension            3.6.6
win-inet-pton                 1.1.0
wrapt                         1.14.1
xarray                        2023.6.0
xlwings                       0.31.4
xyzservices                   2022.9.0
yapf                          0.40.2
yarl                          1.9.3
zict                          3.0.0
zipp                          3.17.0
zope.interface                5.4.0
zstandard                     0.22.0
```

[18]:
```
!pip install tensorflow
```

```
Requirement already satisfied: tensorflow in c:\users\admin\anaconda3\lib\site-
packages (2.18.0)
Requirement already satisfied: tensorflow-intel==2.18.0 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow) (2.18.0)
Requirement already satisfied: absl-py>=1.0.0 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (25.2.10)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (0.2.0)
Requirement already satisfied: libclang>=13.0.0 in
c:\users\admin\anaconda3\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (18.1.1)
```

```
Requirement already satisfied: namex in c:\users\admin\anaconda3\lib\site-
packages (from keras>=3.5.0->tensorflow-intel==2.18.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in c:\users\admin\anaconda3\lib\site-
packages (from keras>=3.5.0->tensorflow-intel==2.18.0->tensorflow) (0.14.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
c:\users\admin\anaconda3\lib\site-packages (from
requests<3,>=2.21.0->tensorflow-intel==2.18.0->tensorflow) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
c:\users\admin\anaconda3\lib\site-packages (from
requests<3,>=2.21.0->tensorflow-intel==2.18.0->tensorflow) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
c:\users\admin\anaconda3\lib\site-packages (from
requests<3,>=2.21.0->tensorflow-intel==2.18.0->tensorflow) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in
c:\users\admin\anaconda3\lib\site-packages (from
requests<3,>=2.21.0->tensorflow-intel==2.18.0->tensorflow) (2024.7.4)
Requirement already satisfied: markdown>=2.6.8 in
c:\users\admin\anaconda3\lib\site-packages (from
tensorboard<2.19,>=2.18->tensorflow-intel==2.18.0->tensorflow) (3.4.1)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in
c:\users\admin\anaconda3\lib\site-packages (from
tensorboard<2.19,>=2.18->tensorflow-intel==2.18.0->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
c:\users\admin\anaconda3\lib\site-packages (from
tensorboard<2.19,>=2.18->tensorflow-intel==2.18.0->tensorflow) (3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in
c:\users\admin\anaconda3\lib\site-packages (from
werkzeug>=1.0.1->tensorboard<2.19,>=2.18->tensorflow-intel==2.18.0->tensorflow)
(2.1.3)
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in
c:\users\admin\anaconda3\lib\site-packages (from rich->keras>=3.5.0->tensorflow-
intel==2.18.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
c:\users\admin\anaconda3\lib\site-packages (from rich->keras>=3.5.0->tensorflow-
intel==2.18.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in c:\users\admin\anaconda3\lib\site-
packages (from markdown-it-py<3.0.0,>=2.2.0->rich->keras>=3.5.0->tensorflow-
intel==2.18.0->tensorflow) (0.1.0)
```

[20]:
```
import numpy as np
import matplotlib.pyplot as  plt
import tensorflow as tf
```

[22]:
```
a = tf.constant(3.)
type(a), a.shape, a.dtype
```

[22]: (tensorflow.python.framework.ops.EagerTensor, TensorShape([]), tf.float32)

[24]:
```
a=tf.constant([3, 2.0,3])
type(a), a.shape, a.dtype
```

[24]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([3]), tf.float32)
```

[28]:
```
a=tf.Variable([[3,2.,3],[3,2.,3]])
type(a), a.shape, a.dtype
```

[28]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 3]),
 tf.float32)
```

[36]:
```
a=tf.Variable([[[3,2.,3]],[[4,2.,4]]])
type(a), a.shape, a.dtype
```

[36]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 1, 3]),
 tf.float32)
```

[38]:
```
a=tf.Variable([[[[[3,2,3],[3,2,3]],]]])
type(a), a.shape, a.dtype
```

[38]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([1, 1, 1, 2, 3]),
 tf.int32)
```

[40]:
```
a = tf.Variable([[[3,2,3],[3,2,3]],[[3,2,3],[3,2,3]]])
type(a), a.shape, a.dtype
```

[40]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 2, 3]),
 tf.int32)
```

Gradient tape

[46]:
```
a=tf.constant([2.])
x=tf.Variable([3.,4.])
y=2*x+a
```

[48]:
```
y.shape,y.dtype,y.numpy()
```

[48]:
```
(TensorShape([2]), tf.float32, array([ 8., 10.], dtype=float32))
```

[50]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y=2*x**2+a
```

[52]:
```
dy_dx = tape.gradient(y,x).numpy()
dy_dx
```

[52]:
```
array([12., 16.], dtype=float32)
```

[54]:
```
a=tf.constant([2.])
x1= tf.Variable([3.,])
x2= tf.Variable([1.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x1)
    tape.watch(x2)

    y=2*x1**2+x2**3+a
```

[56]:
```
dy_dx1, dy_dx2 = tape.gradient(y, [x1,x2])
dy_dx1, dy_dx2
```

[56]:
```
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([12.], dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([3.], dtype=float32)>)
```

[ ]:
```

```

Sigmoid Derivative

[59]:
```
x= tf.Variable([3.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)

    y=1/(1+ tf.exp(-x))

dy_dx = tape.gradient(y,x)
```

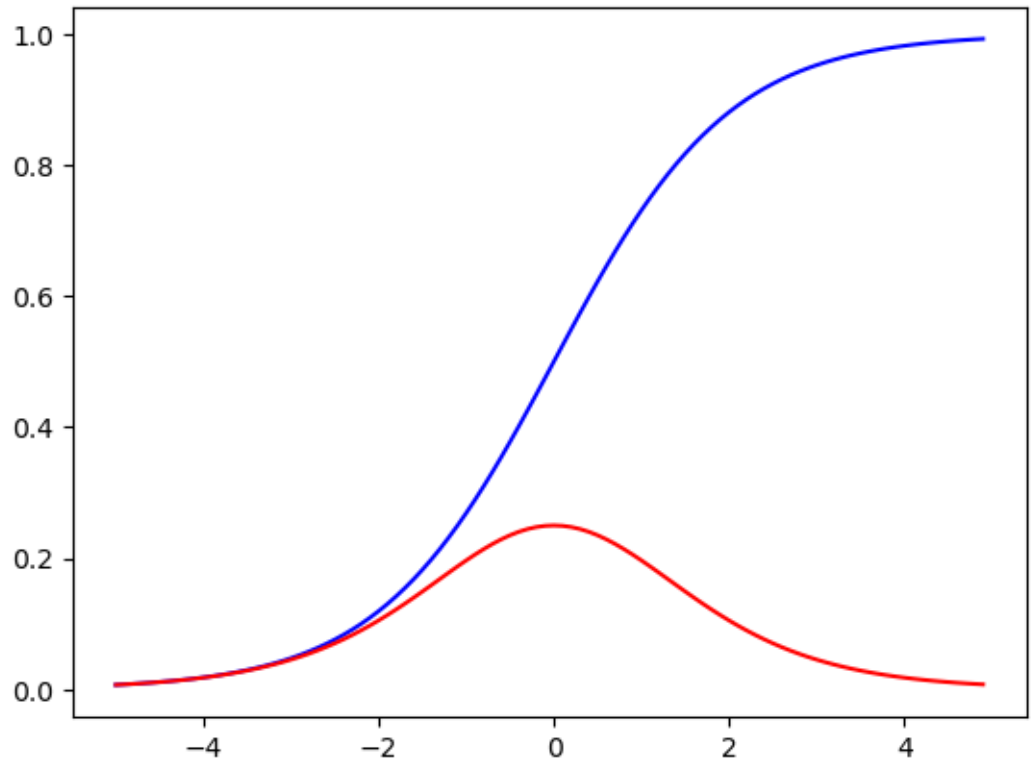
[61]:
```
y, dy_dx
```

[61]:
```
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.95257413],
dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.04517666],
dtype=float32)>)
```

[63]:
```
x= tf.Variable(np.arange(-5,5,0.1))

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y=1/(1+tf.exp(-x))
```

```
dy_dx = tape.gradient(y,x)
```

[65]:
```
plt.plot(x.numpy(), y.numpy(), 'b')
plt.plot(x.numpy(), dy_dx.numpy(), 'r')
plt.show()
```



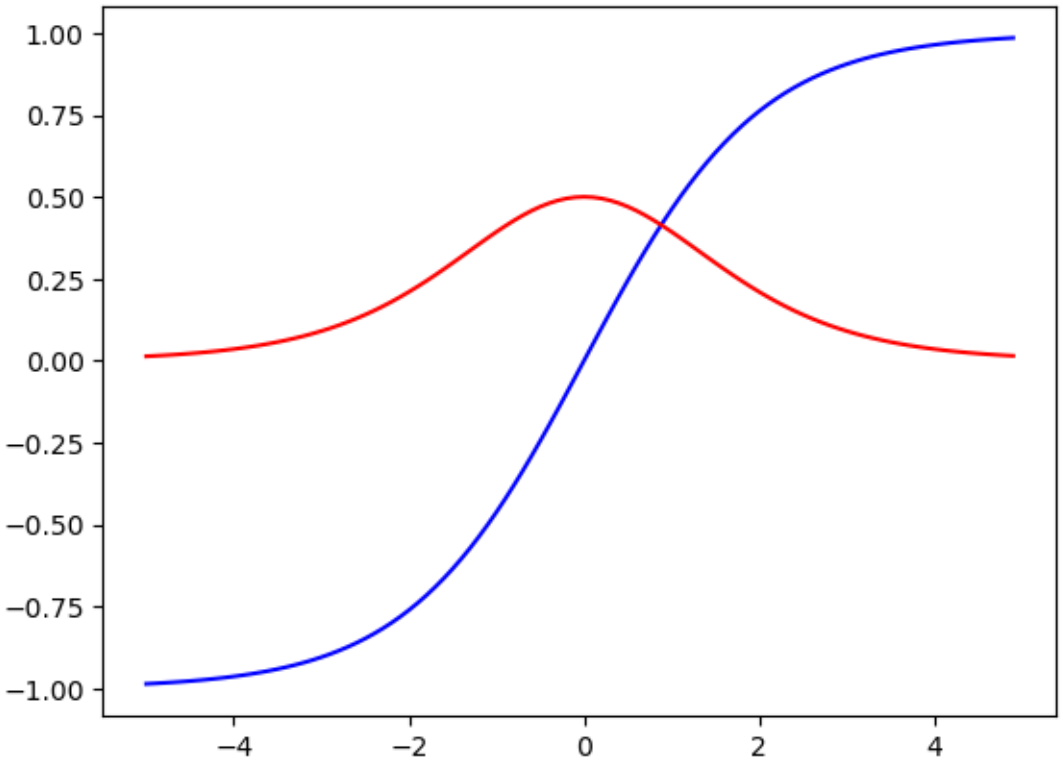
Bipoolar Sigmoidal

[67]:
```
q1 = tf.Variable(np.arange(-5,5,0.1))
```

[71]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(q1)
    y1 = (1 - tf.exp(-q1))/(1+ tf.exp(-q1))
```

[73]:
```
dy_dq1 = tape.gradient(y1,q1)
```

[75]:
```
plt.plot(q1.numpy(), y1.numpy(), 'b')
plt.plot(q1.numpy(), dy_dq1.numpy(), 'r')
plt.show()
```



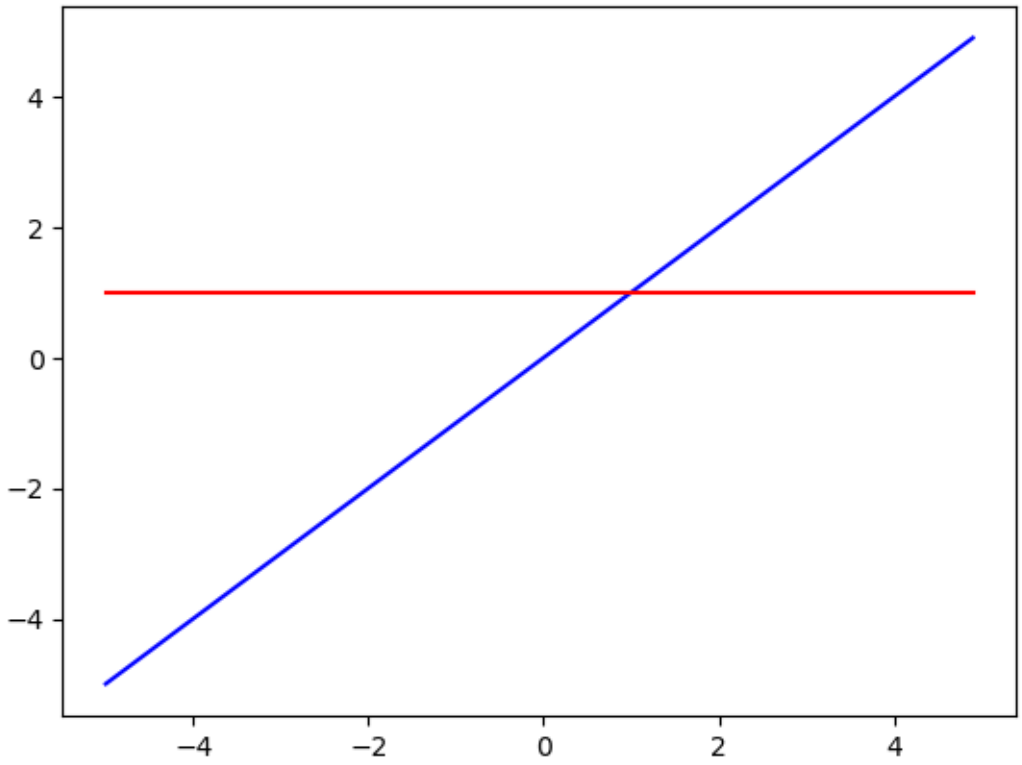
Linear

[85]:
```
l=tf.Variable(np.arange(-5,5,0.1))
```

[89]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(l)
    y2= l
```

[91]:
```
dy_dq1 = tape.gradient(y2,l)
```

[93]:
```
plt.plot(l.numpy(), y2.numpy(), 'b')
plt.plot(l.numpy(), dy_dq1.numpy(), 'r')
plt.show()
```



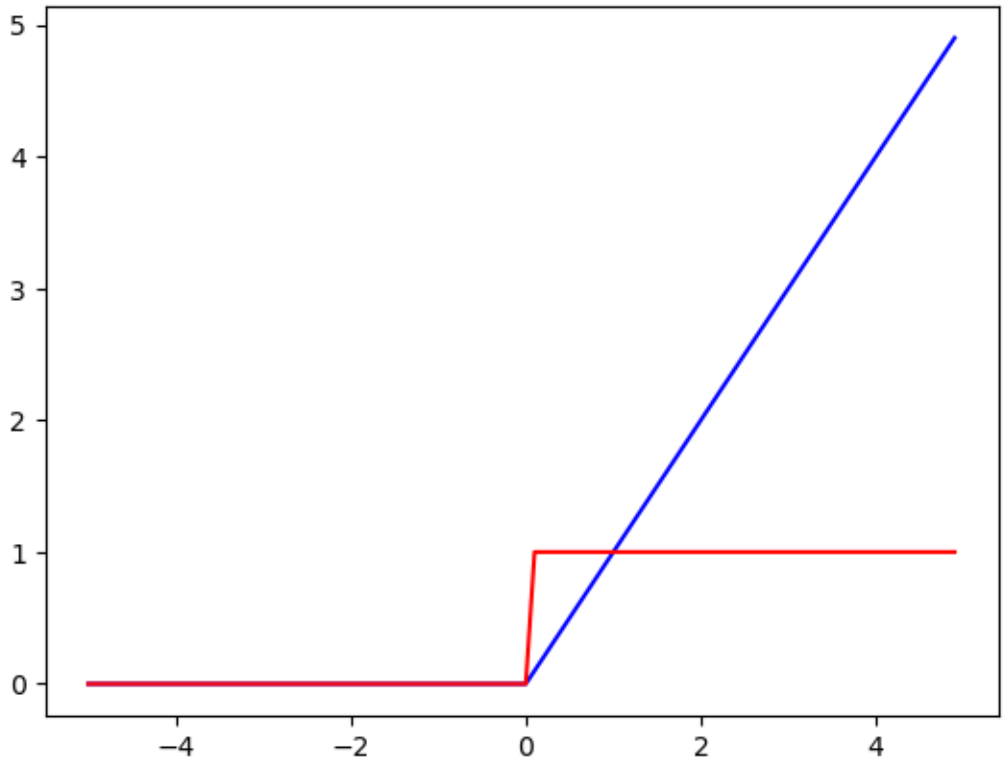
ReLU

[95]:
```
r=tf.Variable(np.arange(-5,5,0.1))
```

[99]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(l)
    y3= tf.keras.activations.relu(r)
```

[101]:
```
dy_dq2 = tape.gradient(y3,r)
```

[103]:
```
plt.plot(r.numpy(), y3.numpy(), 'b')
plt.plot(r.numpy(), dy_dq2.numpy(), 'r')
plt.show()
```

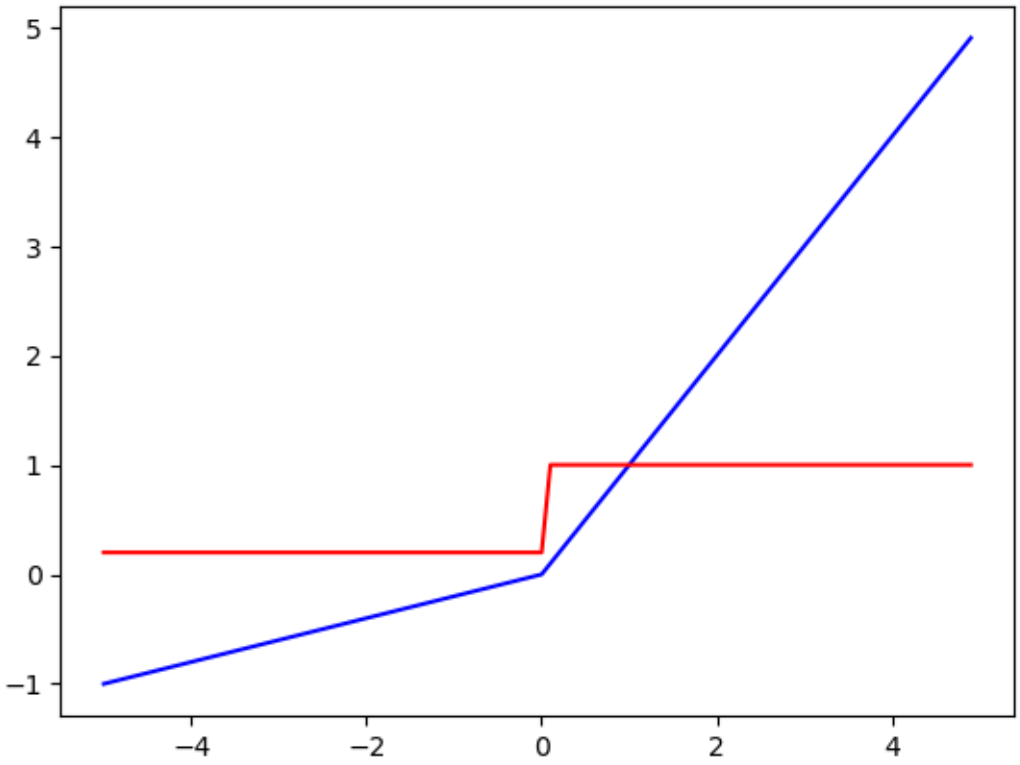


Leaky ReLU

[113]:
```
r=tf.Variable(np.arange(-5,5,0.1))
with tf.GradientTape(persistent=True) as tape:
    tape.watch(l)
    y4= tf.keras.activations.leaky_relu(r)
```

[115]:
```
dy_dq3 = tape.gradient(y4,r)
```

[117]:
```
plt.plot(r.numpy(), y4.numpy(), 'b')
plt.plot(r.numpy(), dy_dq3.numpy(), 'r')
plt.show()
```


5
4
3
2
1
0
−1
−4
−2
0
2
4

[ ]:

mlp-forward-pass-1-layered

April 22, 2025

[84]:
```
import numpy as np
np.set_printoptions(precision = 4)
```

[85]:
```
W_0 = np.array([[-1, -1, -1],
                [0, 1 , -1],
                [-1, 2, 1]], dtype = float)

print(f'{W_0.shape = }')
```

```
W_0.shape = (3, 3)
```

[86]:
```
X = np.array([[0, -1, 1],
              [-1, 0, -1]], dtype = float)
print(f'{X.shape = }')
```

```
X.shape = (2, 3)
```

[87]:
```
f_1 = 'ReLU'
```

[88]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))

def BSigmoid(x):
    retuen (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn = 'Lin'):
    A = np.array(list(map(globals()[fcn], Z)))
    return A
```

0.0.1 FORWARD PASS

[90]:
```
print('Pad bias at top of the input : ')
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(A_0)

print("\n\nInput --> Output")
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end = "\n\n")
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top of the input :
[[ 1.  1.  1.]
 [ 0. -1.  1.]
 [-1.  0. -1.]]


Input --> Output
Z_1 = array([[ 0.,  0., -1.],
       [ 1., -1.,  2.],
       [-2., -3.,  0.]])

A_1 = array([[0., 0., 0.],
       [1., 0., 2.],
       [0., 0., 0.]])
```

0.0.2 SAME WITH SOME MODIFICATIONS

[92]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X

else:
    print('Pad bias at top of the input : ')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(A_0)

print("\n\nInput --> Output")
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end = "\n\n")
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top of the input :
[[ 1.  1.  1.]
 [ 0. -1.  1.]
 [-1.  0. -1.]]
```

```
Input --> Output
Z_1 = array([[ 0.,  0., -1.],
       [ 1., -1.,  2.],
       [-2., -3.,  0.]])

A_1 = array([[0., 0., 0.],
       [1., 0., 2.],
       [0., 0., 0.]])
```

[ ]:

mlp-forward-pass-2-layered

April 22, 2025

[307]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[308]:
```
W_0=np.array([[-1,-1,1],
              [0,1,-1],
              [-1,2,1]],dtype=float)
print(W_0.shape)
```

```
(3, 3)
```

[309]:
```
W_1=np.array([[-1,0,1],
              [1,-1,1],
              [0,-1,1]],dtype=float)
print(W_1.shape)
```

```
(3, 3)
```

[310]:
```
X=np.array([[0,-1,1],
            [-1,0,-1]],dtype=float)
print(X.shape)
```

```
(2, 3)
```

[311]:
```
f_1="ReLU"
f_2="USigmoid"
```

[312]:
```
def USigmoid(x):
    return 1/1+(np.exp(-x))

def BSigmoid(x):
    return (1-np.exp(-x))/(1-np.exp(-x))

def ReLU(x):
    return np.array([max(i,0) for i in x])

def Lin(x):
    return x
```

```
def activation(Z, fcn="Lin"):
    A=np.array(list(map(globals()[fcn],Z)))
    return A
```

[313]:
```
# Forward Pass
```

[314]:
```
if X.shape[0] ==W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0= np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Output')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1,f_1)
print(f'{A_1 = }')
print('\n\nInput -> Output')
if A_1.shape[0] ==W_1.shape[1]:
    print('No bias')

else:
    print('Pad bias at top of input')
    A_1= np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')
Z_2 = W_1 @ A_1
print(f'{Z_2 = }', end='\n\n')
A_2 = activation(Z_2,f_2)
print(f'{A_2 = }')
```

```
Pad bias at top of input
A_0 = array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])


Input -> Output
Z_1 = array([[-2.,  0., -3.],
       [ 1., -1.,  2.],
       [-2., -3.,  0.]])

A_1 = array([[0., 0., 0.],
       [1., 0., 2.],
       [0., 0., 0.]])
```

```
Input -> Output
No bias
A_0 = array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])
Z_2 = array([[ 0.,  0.,  0.],
       [-1.,  0., -2.],
       [-1.,  0., -2.]])

A_2 = array([[2.    , 2.    , 2.    ],
       [3.7183, 2.    , 8.3891],
       [3.7183, 2.    , 8.3891]])
```

[ ]:

[ ]:

```
rfc3986-validator          0.1.1
rpds-py                    0.22.3
Send2Trash                 1.8.2
setuptools                 75.1.0
sip                        6.7.12
six                        1.16.0
sniffio                    1.3.0
soupsieve                  2.5
stack-data                 0.2.0
terminado                  0.17.1
tinycss2                   1.2.1
tornado                    6.4.2
traitlets                  5.14.3
typing_extensions          4.12.2
urllib3                    2.2.3
wcwidth                    0.2.5
webencodings               0.5.1
websocket-client           1.8.0
wheel                      0.44.0
widgetsnbextension         4.0.13
win-inet-pton              1.1.0
```

[2]:
```
!pip install tensorflow
```

```
Collecting tensorflow
  Downloading tensorflow-2.18.0-cp311-cp311-win_amd64.whl.metadata (3.3 kB)
Collecting tensorflow-intel==2.18.0 (from tensorflow)
  Downloading tensorflow_intel-2.18.0-cp311-cp311-win_amd64.whl.metadata (4.9
kB)
Collecting absl-py>=1.0.0 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading absl_py-2.1.0-py3-none-any.whl.metadata (2.3 kB)
Collecting astunparse>=1.6.0 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading flatbuffers-25.2.10-py2.py3-none-any.whl.metadata (875 bytes)
Collecting gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 (from tensorflow-
intel==2.18.0->tensorflow)
  Downloading gast-0.6.0-py3-none-any.whl.metadata (1.3 kB)
Collecting google-pasta>=0.1.1 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading google_pasta-0.2.0-py3-none-any.whl.metadata (814 bytes)
Collecting libclang>=13.0.0 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading libclang-18.1.1-py2.py3-none-win_amd64.whl.metadata (5.3 kB)
Collecting opt-einsum>=2.3.2 (from tensorflow-intel==2.18.0->tensorflow)
  Downloading opt_einsum-3.4.0-py3-none-any.whl.metadata (6.3 kB)
Requirement already satisfied: packaging in
c:\users\admin\.conda\envs\sclab\lib\site-packages (from tensorflow-
intel==2.18.0->tensorflow) (24.2)
Collecting
```

```
Downloading termcolor-2.5.0-py3-none-any.whl (7.8 kB)
Downloading wrapt-1.17.2-cp311-cp311-win_amd64.whl (38 kB)
Downloading Markdown-3.7-py3-none-any.whl (106 kB)
Downloading tensorboard_data_server-0.7.2-py3-none-any.whl (2.4 kB)
Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
Downloading namex-0.0.8-py3-none-any.whl (5.8 kB)
Downloading optree-0.14.0-cp311-cp311-win_amd64.whl (300 kB)
Downloading rich-13.9.4-py3-none-any.whl (242 kB)
Downloading markdown_it_py-3.0.0-py3-none-any.whl (87 kB)
Downloading mdurl-0.1.2-py3-none-any.whl (10.0 kB)
Installing collected packages: namex, libclang, flatbuffers, wrapt, werkzeug,
termcolor, tensorflow-io-gcs-filesystem, tensorboard-data-server, protobuf,
optree, opt-einsum, numpy, mdurl, markdown, grpcio, google-pasta, gast,
astunparse, absl-py, tensorboard, ml-dtypes, markdown-it-py, h5py, rich, keras,
tensorflow-intel, tensorflow
  Attempting uninstall: numpy
    Found existing installation: numpy 2.2.1
    Uninstalling numpy-2.2.1:
      Successfully uninstalled numpy-2.2.1
Successfully installed absl-py-2.1.0 astunparse-1.6.3 flatbuffers-25.2.10
gast-0.6.0 google-pasta-0.2.0 grpcio-1.70.0 h5py-3.13.0 keras-3.8.0
libclang-18.1.1 markdown-3.7 markdown-it-py-3.0.0 mdurl-0.1.2 ml-dtypes-0.4.1
namex-0.0.8 numpy-2.0.2 opt-einsum-3.4.0 optree-0.14.0 protobuf-5.29.3
rich-13.9.4 tensorboard-2.18.0 tensorboard-data-server-0.7.2 tensorflow-2.18.0
tensorflow-intel-2.18.0 tensorflow-io-gcs-filesystem-0.31.0 termcolor-2.5.0
werkzeug-3.1.3 wrapt-1.17.2
```

[3]:
```
!pip install matplotlib
```

```
Collecting matplotlib
  Downloading matplotlib-3.10.0-cp311-cp311-win_amd64.whl.metadata (11 kB)
Collecting contourpy>=1.0.1 (from matplotlib)
  Downloading contourpy-1.3.1-cp311-cp311-win_amd64.whl.metadata (5.4 kB)
Collecting cycler>=0.10 (from matplotlib)
  Using cached cycler-0.12.1-py3-none-any.whl.metadata (3.8 kB)
Collecting fonttools>=4.22.0 (from matplotlib)
  Downloading fonttools-4.56.0-cp311-cp311-win_amd64.whl.metadata (103 kB)
Collecting kiwisolver>=1.3.1 (from matplotlib)
  Downloading kiwisolver-1.4.8-cp311-cp311-win_amd64.whl.metadata (6.3 kB)
Requirement already satisfied: numpy>=1.23 in
c:\users\admin\.conda\envs\sclab\lib\site-packages (from matplotlib) (2.0.2)
Requirement already satisfied: packaging>=20.0 in
c:\users\admin\.conda\envs\sclab\lib\site-packages (from matplotlib) (24.2)
Collecting pillow>=8 (from matplotlib)
  Downloading pillow-11.1.0-cp311-cp311-win_amd64.whl.metadata (9.3 kB)
Collecting pyparsing>=2.3.1 (from matplotlib)
  Downloading pyparsing-3.2.1-py3-none-any.whl.metadata (5.0 kB)
Requirement already satisfied: python-dateutil>=2.7 in
```

```
c:\users\admin\.conda\envs\sclab\lib\site-packages (from matplotlib)
(2.9.0.post0)
Requirement already satisfied: six>=1.5 in
c:\users\admin\.conda\envs\sclab\lib\site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)
Downloading matplotlib-3.10.0-cp311-cp311-win_amd64.whl (8.0 MB)
   ---------------------------------------- 0.0/8.0 MB ? eta -:--:--
   -------------- ------------------------- 3.1/8.0 MB 20.5 MB/s eta 0:00:01
   ---------------------------------------- 8.0/8.0 MB 21.6 MB/s eta 0:00:00
Downloading contourpy-1.3.1-cp311-cp311-win_amd64.whl (219 kB)
Using cached cycler-0.12.1-py3-none-any.whl (8.3 kB)
Downloading fonttools-4.56.0-cp311-cp311-win_amd64.whl (2.2 MB)
   ---------------------------------------- 0.0/2.2 MB ? eta -:--:--
   ---------------------------------------- 2.2/2.2 MB 62.5 MB/s eta 0:00:00
Downloading kiwisolver-1.4.8-cp311-cp311-win_amd64.whl (71 kB)
Downloading pillow-11.1.0-cp311-cp311-win_amd64.whl (2.6 MB)
   ---------------------------------------- 0.0/2.6 MB ? eta -:--:--
   ---------------------------------------- 2.6/2.6 MB 50.3 MB/s eta 0:00:00
Downloading pyparsing-3.2.1-py3-none-any.whl (107 kB)
Installing collected packages: pyparsing, pillow, kiwisolver, fonttools, cycler,
contourpy, matplotlib
Successfully installed contourpy-1.3.1 cycler-0.12.1 fonttools-4.56.0
kiwisolver-1.4.8 matplotlib-3.10.0 pillow-11.1.0 pyparsing-3.2.1
```

Load library

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

[21]:
```
a = tf.constant(3e-38)
type(a), a.shape, a.dtype
```

[21]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([]), tf.float32)
```

[7]:
```
a = tf.constant([3, 2, 3.])
type(a), a.shape, a.dtype
```

[7]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([3]), tf.float32)
```

[22]:
```
a = tf.Variable([[3, 2., 3], [3, 2, 3]])
type(a), a.shape, a.dtype
```

[22]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 3]),
 tf.float32)
```

[26]:
```
a = tf.Variable([[[[[3, 2, 3], [3, 2, 3]],]]])
type(a), a.shape, a.dtype
```

[26]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([1, 1, 1, 2, 3]),
 tf.int32)
```

[13]:
```
a = tf.Variable([[[3, 2, 3], [3, 2, 3]],[[3, 2, 3], [3, 2, 3]]])
type(a), a.shape, a.dtype
```

[13]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 2, 3]),
 tf.int32)
```

Gradient tape

[34]:
```
a = tf.constant([2.])
x = tf.Variable([3.,4])
y = 2*x + a
```

[35]:
```
y.shape, y.dtype, y.numpy()
```

[35]:
```
(TensorShape([2]), tf.float32, array([ 8., 10.], dtype=float32))
```

[41]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = 2*x**1 + a
```

[42]:
```
dy_dx = tape.gradient(y, x).numpy()
dy_dx
```

[42]:
```
array([2., 2.], dtype=float32)
```

[43]:
```
a = tf.constant([2.])
w1 = tf.Variable([3.,])
w2 = tf.Variable([1.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(w1)
    tape.watch(w2)

    y = 2*w1**2 + w2**3 + a
```

[45]:
```
dy_dw1, dy_dw2 = tape.gradient(y, [w1, w2])
dy_dw1.numpy(), dy_dw2.numpy()
```

[45]:
```
(array([12.], dtype=float32), array([3.], dtype=float32))
```

[46]:
```
a = tf.constant([2.])
W = tf.Variable([3., 2.])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(W)

    y = 2*W[0]**2 + W[1]**3 + a
```

[48]:
```
tape.gradient(y, W)
```

[48]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([12., 12.], dtype=float32)>
```

Sigmoid derivative

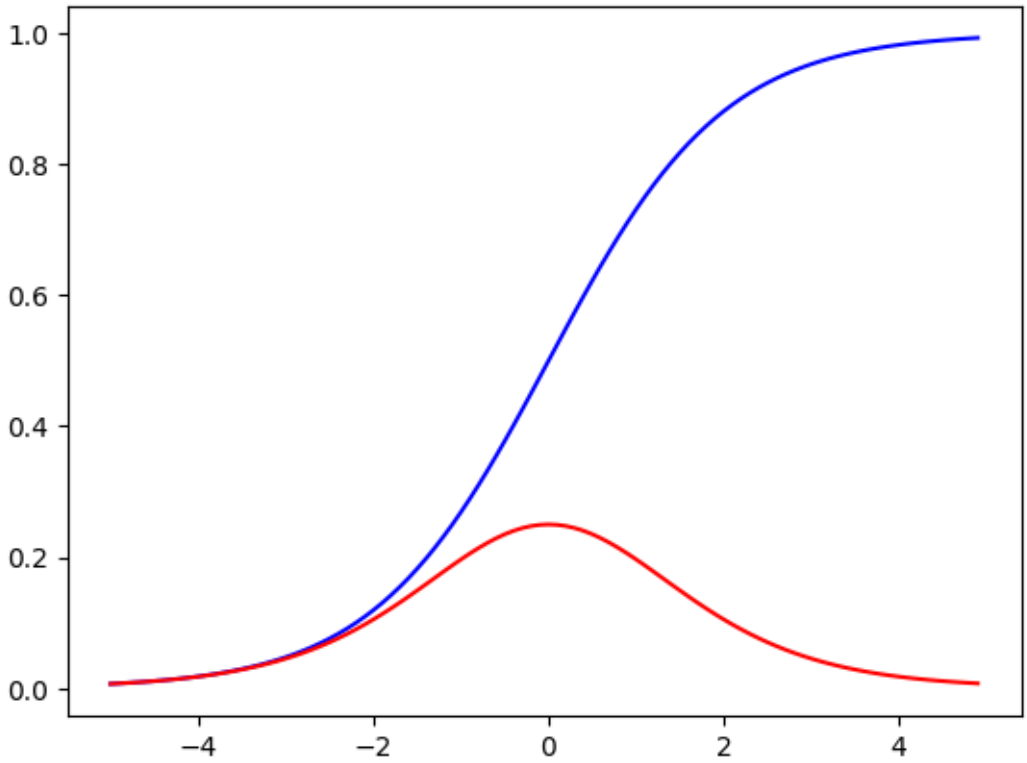
[22]:
```
x = tf.Variable([3.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)

    y = 1/(1 + tf.exp(-x))

dy_dx = tape.gradient(y, x)
```

[23]:
```
y, dy_dx
```

[23]:
```
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.95257413],
dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.04517666],
dtype=float32)>)
```

[49]:
```
x = tf.Variable(np.arange(-5, 5, 0.1))

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = 1/(1 + tf.exp(-x))

dy_dx = tape.gradient(y, x)
```

[50]:
```
plt.plot(x.numpy(), y.numpy(), 'b')
plt.plot(x.numpy(), dy_dx.numpy(), 'r')
plt.show()
```



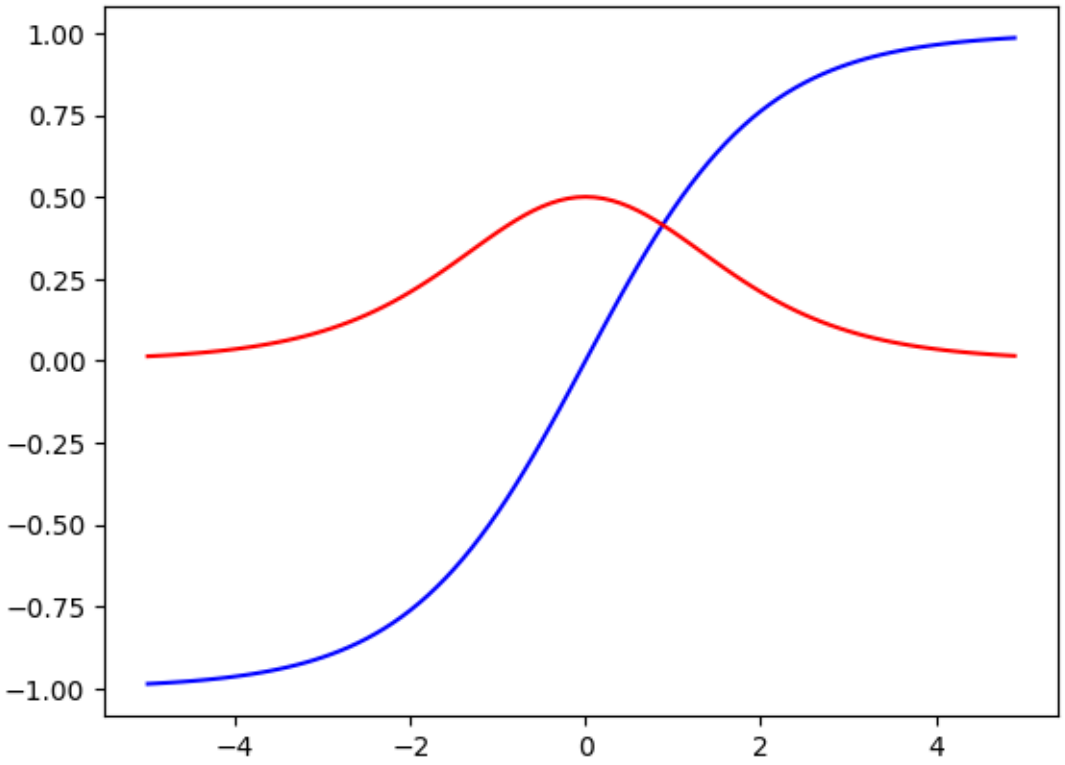
Bipoolar Sigmoidal

[30]:
```
q1 = tf.Variable(np.arange(-5,5,0.1))
```

[31]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(q1)
    y1 = (1 - tf.exp(-q1))/(1 + tf.exp(-q1))
```

[32]:
```
dy_dq1 =  tape.gradient(y1,q1)
```

[33]:
```
plt.plot(q1.numpy(), y1.numpy(), 'b')
plt.plot(q1.numpy(), dy_dq1.numpy(), 'r')
plt.show()
```



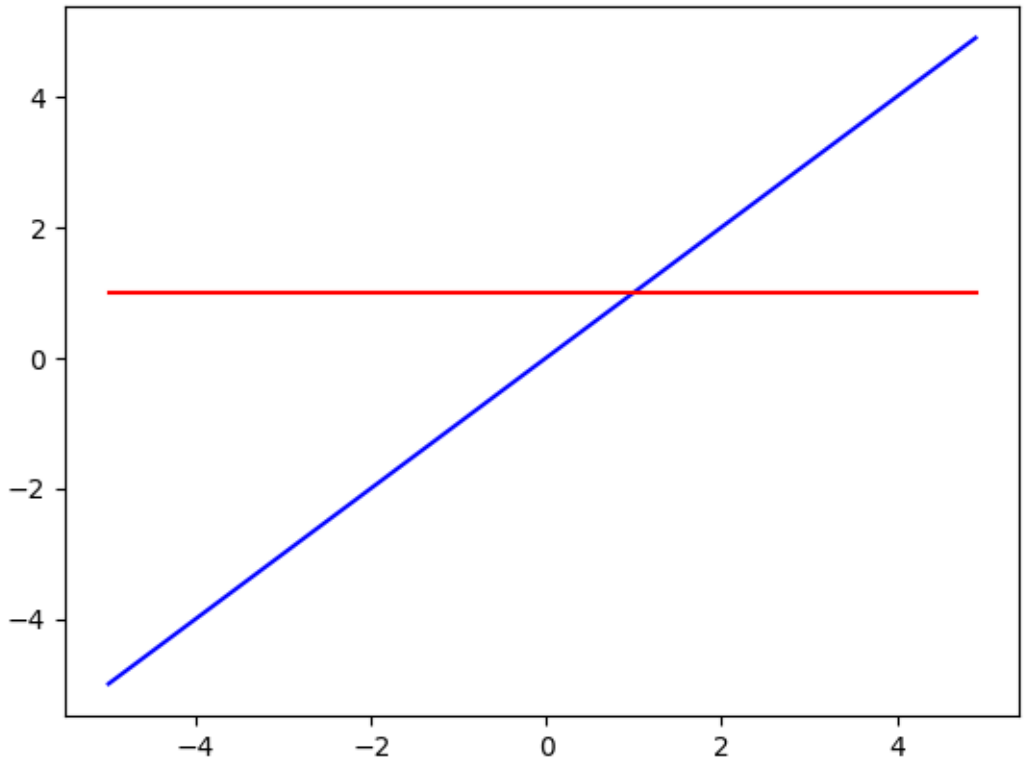
Linear

[114]:
```
l = tf.Variable(np.arange(-5,5,0.1))
```

[115]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(l)
    y2 = l
```

[116]:
```
dy_dq1 =  tape.gradient(y2,l)
```

[117]:
```
plt.plot(l.numpy(), y2.numpy(), 'b')
plt.plot(l.numpy(), dy_dq1.numpy(), 'r')
plt.show()
```

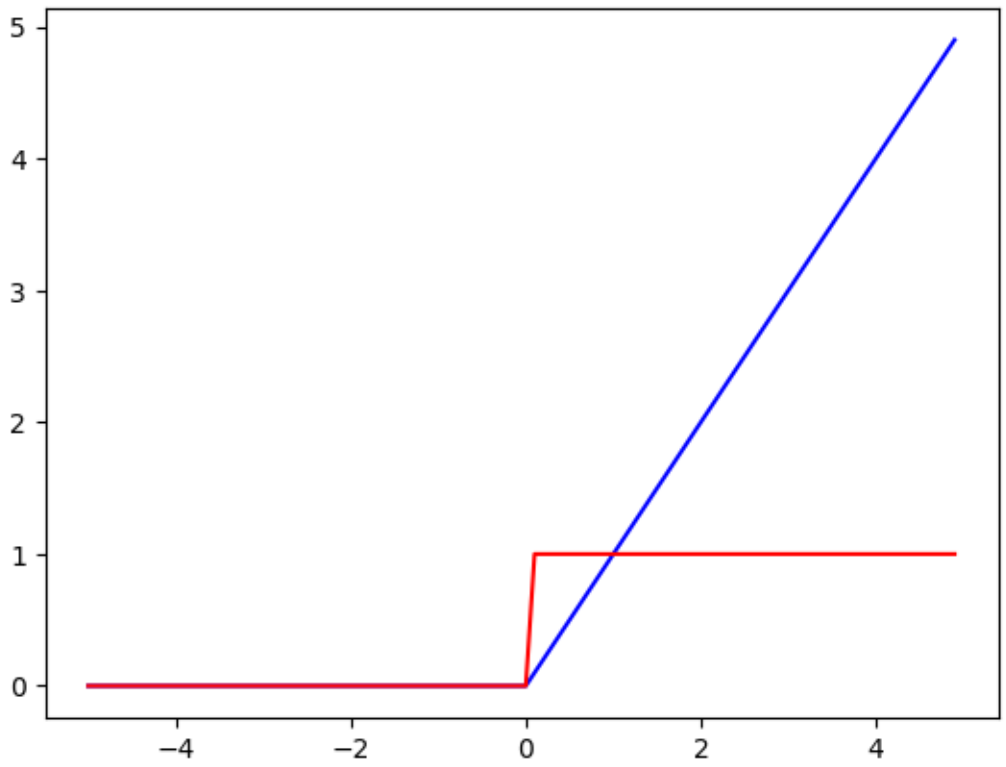


[118]:
```
r = tf.Variable(np.arange(-5,5,0.1))
```

[120]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(l)
    y3 = tf.keras.activations.relu(r)
```

[121]:
```
dy_dq2 =  tape.gradient(y3,r)
```

[123]:
```
plt.plot(r.numpy(), y3.numpy(), 'b')
plt.plot(r.numpy(), dy_dq2.numpy(), 'r')
plt.show()
```


5
4
3
2
1
0
−4
−2
0
2
4

[ ]:

mlp-forward-pass-1-layered

April 22, 2025

[1]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[23]:
```
W_0=np.array([[-1,-1,1],
              [0,1,-1],
              [-1,2,1]],dtype=float)
print(W_0.shape)
```

```
(3, 3)
```

[25]:
```
X=np.array([[0,-1,1],
            [-1,0,-1]],dtype=float)
print(X.shape)
```

```
(2, 3)
```

[27]:
```
f_1="USigmoid"
```

[29]:
```
def USigmoid(x):
    return 1/1+(np.exp(-x))

def BSigmoid(x):
    return (1-np.exp(-x))/(1-np.exp(-x))

def ReLU(x):
    return np.array([max(i,0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn="Lin"):
    A=np.array(list(map(globals()[fcn],Z)))
    return A
```

[31]:
```
# Forward Pass
```

[33]:
```
if X.shape[0] ==W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0= np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Output')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1,f_1)
print(f'{A_1 = }')
```

```
Pad bias at top of input
A_0 = array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])


Input -> Output
Z_1 = array([[-2.,  0., -3.],
       [ 1., -1.,  2.],
       [-2., -3.,  0.]])

A_1 = array([[ 8.3891,  2.    , 21.0855],
       [ 1.3679,  3.7183,  1.1353],
       [ 8.3891, 21.0855,  2.    ]])
```

[ ]:

[ ]:

mlp-backward-pass-1-layered-1

April 22, 2025

[103]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[104]:
```
W_0=np.array([[1,1,1],], dtype=float)
print(W_0.shape)
```

```
(1, 3)
```

[105]:
```
t=np.array([[0.5,1],], dtype=float)
```

[106]:
```
X=np.array([[1,1,1],
            [3,3,3]],dtype=float)
print(X.shape)
```

```
(2, 3)
```

[ ]:
```

```

[107]:
```
f_1="USigmoid"
```

[108]:
```
lr = 1
```

[109]:
```
def USigmoid(x, direction):
    if direction == 'F':
        return 1/1+(np.exp(-x))
    else:
        return USigmoid(x, 'F')*(1-USigmoid(x,'F'))

def BSigmoid(x, direction):
    if direction == 'F':
        return (1-np.exp(-x))/(1-np.exp(-x))
    else:
        return 0.5*(1-BSigmoid(x, 'F')**2)

def ReLU(x, direction):
    if direction =='F':
        return max(x,0)
    else:
```

```
        return float(x>0)

def Lin(x, direction):
    if direction == 'F':
        return x
    else:
        return 1

def activation(Z, fcn="Lin", direction ='F'):
    A=np.array(list(map(globals()[fcn],Z , np.repeat(direction, Z.shape[0]))))
    return A
```

[110]:
```
MAX_EPOCH = 1
```

[111]:
```
print ('Pad bias at top of input')
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(A_0)
for ep in range(MAX_EPOCH):
    print('\nEPOCH-', ep+1, '='*80)
    for itr, (x,y) in enumerate(zip(A_0.T, t.T)):
        print('\nITER-', itr+1, '-'*80)
        ### Forward pass
        print('Input -> Output')
        Z_1 = W_0 @ x
        print(f'{Z_1 = }')
        A_1 = activation(Z_1, f_1)
        print(f'{A_1 = }')


        ### Backward pass
        print('\nOutput -> Input')
        Error = 0.5*(A_1 - t)**2
        print(f'{Error = }')
        dE_dW = (A_1 - y) * activation (Z_1 , f_1, 'B') * x
        print(f'{dE_dW = }')
        W_0 = W_0 - lr*dE_dW
        print(f'{W_0 = }')
```

```
Pad bias at top of input
[[1. 1. 1.]
 [1. 1. 1.]
 [3. 3. 3.]]

EPOCH- 1
================================================================================

ITER- 1
```

```
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.])
A_1 = array([1.0067])

Output -> Input
Error = array([[1.2839e-01, 2.2700e-05]])
dE_dW = array([-0.0034, -0.0034, -0.0103])
W_0 = array([[1.0034, 1.0034, 1.0103]])

ITER- 2
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.0378])
A_1 = array([1.0065])

Output -> Input
Error = array([[1.2827e-01, 2.1047e-05]])
dE_dW = array([-4.2366e-05, -4.2366e-05, -1.2710e-04])
W_0 = array([[1.0035, 1.0035, 1.0104]])
```

[ ]:

[ ]:

[ ]:

[ ]:

mlp-backward-pass-1-layered

April 22, 2025

[10]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[11]:
```
W_0 = np.array([[1, 1, 1],], dtype=float)
```

[12]:
```
X = np.array([[1, 1, 1],
              [3, 3, 3]], dtype=float)
```

[13]:
```
t = np.array([[1, 1, 1],], dtype=float)
```

[14]:
```
f_1 = 'USigmoid'
```

[15]:
```
lr = 1
```

[16]:
```
def USigmoid(x, direction):
    if direction == 'F':
        return 1/(1 + np.exp(-x))
    else:
        return USigmoid(x, 'F')*(1-USigmoid(x, 'F'))

def BSigmoid(x, direction):
    if direction == 'F':
        return (1 - np.exp(-x))/(1 + np.exp(-x))
    else:
        return 0.5*(1-BSigmoid(x, 'F')**2)

def ReLU(x, direction):
    if direction == 'F':
        return max(x, 0)
    else:
        return float(x > 0)

def Lin(x, direction):
    if direction == 'F':
        return x
    else:
        return 1
```

```
def activation(Z, fcn='Lin', direction = 'F'):
    A = np.array(list(map(globals()[fcn], Z, np.repeat(direction, Z.shape[0]))))
    return A
```

[17]:
```
MAX_EPOCHS = 1
```

[18]:
```
print('Pad bias at top of input')
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(A_0)
for ep in range(MAX_EPOCHS):
    print('\nEPOCH-', ep+1, '='*80)
    for itr, (x, y) in enumerate(zip(A_0.T, t.T)):
        print('\nITER-', itr+1, '-'*80)
        ### Forward pass
        print('Input -> Output')
        Z_1 = W_0 @ x
        print(f'{Z_1 = }')
        A_1 = activation(Z_1, f_1)
        print(f'{A_1 = }')

        ### Backward pass
        print('\nOutput -> Input')
        Error = 0.5*(A_1 - t)**2
        print(f'{Error = }')
        dE_dw = (A_1 - y) * activation(Z_1, f_1, 'B') * x
        print(f'{dE_dw = }')
        W_0 = W_0 - lr*dE_dw
        print(f'{W_0 = }')
```

```
Pad bias at top of input
[[1. 1. 1.]
 [1. 1. 1.]
 [3. 3. 3.]]

EPOCH- 1
================================================================================

ITER- 1
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.])
A_1 = array([0.9933])

Output -> Input
Error = array([[2.2397e-05, 2.2397e-05, 2.2397e-05]])
dE_dw = array([-4.4494e-05, -4.4494e-05, -1.3348e-04])
```

```
W_0 = array([[1.    , 1.    , 1.0001]])

ITER- 2
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.0005])
A_1 = array([0.9933])

Output -> Input
Error = array([[2.2375e-05, 2.2375e-05, 2.2375e-05]])
dE_dw = array([-4.4451e-05, -4.4451e-05, -1.3335e-04])
W_0 = array([[1.0001, 1.0001, 1.0003]])

ITER- 3
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.001])
A_1 = array([0.9933])

Output -> Input
Error = array([[2.2354e-05, 2.2354e-05, 2.2354e-05]])
dE_dw = array([-4.4408e-05, -4.4408e-05, -1.3323e-04])
W_0 = array([[1.0001, 1.0001, 1.0004]])
```

mlp-forward-pass-2-layered

April 22, 2025

[8]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[9]:
```
W_0 = np.array([[1, -1],
                [1, -1]], dtype=float)
print(f'{W_0.shape=}')
```

```
W_0.shape=(2, 2)
```

[10]:
```
W_1 = np.array([[-1, 1, 0,],
                [0, -1, 1],
                [1, 0, -1]], dtype=float)
print(f'{W_1.shape=}')
```

```
W_1.shape=(3, 3)
```

[11]:
```
X = np.array([[1, 1, 0],
              [0, 1, 1]], dtype=float)
print(f'{X.shape=}')
```

```
X.shape=(2, 3)
```

[12]:
```
f_1 = 'Lin'
f_2 = 'ReLU'
```

[13]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))

def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn='Lin'):
```

```
    A = np.array(list(map(globals()[fcn], Z)))
    return A
```

0.0.1 Forward pass

[14]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Hidden')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')

print('\n\nHidden -> Output')
if A_1.shape[0] == W_1.shape[1]:
    print('No bias')
else:
    print('Pad bias at top of A1')
    A_1 = np.vstack((np.ones((1, A_1.shape[1])), A_1))
print(f'{A_1 = }')

Z_2 = W_1 @ A_1
print(f'{Z_2 = }', end='\n\n')
A_2 = activation(Z_2, f_2)
print(f'{A_2 = }')
```

```
No bias
A_0 = array([[1., 1., 0.],
       [0., 1., 1.]])


Input -> Hidden
Z_1 = array([[ 1.,  0., -1.],
       [ 1.,  0., -1.]])

A_1 = array([[ 1.,  0., -1.],
       [ 1.,  0., -1.]])


Hidden -> Output
Pad bias at top of A1
```

```
A_1 = array([[ 1.,  1.,  1.],
       [ 1.,  0., -1.],
       [ 1.,  0., -1.]])
Z_2 = array([[ 0., -1., -2.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  2.]])

A_2 = array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 1., 2.]])
```

[ ]:

single-layer-perceptron

April 22, 2025

[156]:
```
import numpy as np
```

0.0.1 Input

[158]:
```
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
print(f'{X.shape=}')
X
```

```
X.shape=(2, 4)
```

[158]:
```
array([[0, 0, 1, 1],
       [0, 1, 0, 1]])
```

[159]:
```
t = np.array([[0, 0, 0, 1]])
print(f'{t.shape=}')
t
```

```
t.shape=(1, 4)
```

[159]:
```
array([[0, 1, 1, 1]])
```

0.0.2 Add bias

[161]:
```
A = np.vstack((np.ones((1, 4)), X))
A
```

[161]:
```
array([[1., 1., 1., 1.],
       [0., 0., 1., 1.],
       [0., 1., 0., 1.]])
```

0.0.3 Initialize weights

[163]:
```
W = np.array([[0, 0, 0]])
print(f'{W.shape=}')
```

```
W.shape=(1, 3)
```

[164]:
```
MAX_EPOCH = 10
```

0.0.4 Perceptron learning

[166]:
```
for ep in range(MAX_EPOCH):
    print('Epoch-', ep+1, '='*50)
    isConv = True
    for x, y in zip(A.T, t.ravel()):
        Z = W @ x.T
        # Actually from positive but predicted as negative
        if y>0 and Z < 0:
            W = W + x
            isConv = False
            print('Update: W+x')
        # Actually from negative but predicted as positive
        if y<1 and Z >= 0:
            W = W - x
            isConv = False
            print('Update: W-x')
        print(x, y, Z, W)
    if isConv:
        print('Final weights: ', W)
        break
```

```
Epoch- 1 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  0.  0.]]
Update: W+x
[1. 0. 1.] 1 [-1.] [[0. 0. 1.]]
[1. 1. 0.] 1 [0.] [[0. 0. 1.]]
[1. 1. 1.] 1 [1.] [[0. 0. 1.]]
Epoch- 2 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  0.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  0.  1.]]
Update: W+x
[1. 1. 0.] 1 [-1.] [[0. 1. 1.]]
[1. 1. 1.] 1 [2.] [[0. 1. 1.]]
Epoch- 3 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  1.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 0.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 1.] 1 [1.] [[-1.  1.  1.]]
Epoch- 4 ==================================================
[1. 0. 0.] 0 [-1.] [[-1.  1.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  1.  1.]]
```

```
[1. 1. 0.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 1.] 1 [1.] [[-1.  1.  1.]]
Final weights:  [[-1.  1.  1.]]
```

[ ]:

[167]:
```
Z = W @ A
y_pred = [int(pred >= 0) for pred in Z.ravel()]
print(f'{y_pred=}')
```

```
y_pred=[0, 1, 1, 1]
```

[ ]:

exp-3-layer-2

April 22, 2025

[1]:
```
import numpy as np
np.set_printoptions(precision = 4)
```

[2]:
```
W_0 = np.array ([[1, -1, 1],
                 [0, 1, 1],
                 [-1, 2, 1]], dtype= float)
print(f'{W_0.shape = }')
```

```
W_0.shape = (3, 3)
```

[17]:
```
W_1 = np.array ([[-1, -1, -1],
                 [0, 1, -1],
                 [-1, 2, 1]], dtype= float)
print(f'{W_1.shape = }')
```

```
W_1.shape = (3, 3)
```

[18]:
```
X = np.array ([[0, -1, 1],
               [-1, 0, -1]], dtype= float)
```

[19]:
```
f_1 = 'ReLU'
```

[20]:
```
f_2 = 'BSigmoid'
```

[21]:
```
def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])
```

[22]:
```
def Lin (X):
    return X
```

[23]:
```
def activation(Z, fcn='Lin'):
    A= np.array(list(map(globals()[fcn],Z)))
    return A
```

[15]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 =}')

print('\ninput > output')
Z_1 = W_0@A_0
print(f'{Z_1 = }', end = '\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top input
A_0 =array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])

input > output
Z_1 = array([[ 0.,  2., -1.],
       [-1., -1.,  0.],
       [-2., -3.,  0.]])

A_1 = array([[0., 2., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[16]:
```
if X.shape[0] == W_1.shape[1]:
    print('No bias')
    A_1 = X
else:
    print('Pad bias at top input')
    A_1 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_1 =}')

print('\ninput > output')
Z_2 = W_1@A_1
print(f'{Z_2 = }', end = '\n\n')



A_2= activation(Z_2, f_2)

print(f'{A_2 = }')
```

```
Pad bias at top input
A_1 =array([[ 1.,  1.,  1.],
```

```
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])

input > output
Z_2 = array([[ 0.,  2., -1.],
       [-1., -1.,  0.],
       [-2., -3.,  0.]])

A_2 = array([[ 0.    ,  0.7616, -0.4621],
       [-0.4621, -0.4621,  0.    ],
       [-0.7616, -0.9051,  0.    ]])
```

[ ]:

[ ]:

mlp-forward-pass-1-layered

April 22, 2025

[13]:
```
import numpy as np
np.set_printoptions(precision=4)
```

[14]:
```
W_0 = np.array([[-1, -1, 1],
                [-1, 2, 1]], dtype=float)
print(f'{W_0.shape=}')
```

```
W_0.shape=(2, 3)
```

[15]:
```
X = np.array([[0, -1, 1],
              [-1, 0, -1]], dtype=float)
print(f'{X.shape=}')
```

```
X.shape=(2, 3)
```

[16]:
```
f_1 = 'USigmoid'
```

[17]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))

def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn='Lin'):
    A = np.array(list(map(globals()[fcn], Z)))
    return A
```

0.0.1 Forward pass

[18]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Output')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top of input
A_0 = array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])


Input -> Output
Z_1 = array([[-2.,  0., -3.],
       [-2., -3.,  0.]])

A_1 = array([[0.1192, 0.5   , 0.0474],
       [0.1192, 0.0474, 0.5   ]])
```

[ ]:

exp-3-layer-1

April 22, 2025

0.0.1 single layer perceptron forward pass

[ ]:
```
import numpy as np
np.set_printoptions(precision = 4)
```

[27]:
```
W_0 = np.array ([[1, -1, 1],
                 [0, 1, 1],
                 [-1, 2, 1]], dtype= float)
print(f'{W_0.shape = }')
```

```
W_0.shape = (3, 3)
```

[11]:
```
X = np.array ([[0, -1, 1],
               [-1, 0, -1]], dtype= float)
```

[13]:
```
f_1 = 'ReLU'
```

[14]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))
```

[15]:
```
def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))
```

[16]:
```
def ReLU(X):
    return np.array([max(i,0) for i in X])
```

[19]:
```
def Lin (X):
    return X
```

[22]:
```
def activation(Z, fcn='Lin'):
    A= np.array(list(map(globals()[fcn],Z)))
    return A
```

1 Forward Pass

[31]:
```
print('Pad bias at top input')
A_0 = np.vstack((np.ones((1, X.shape[1])), X))

print(A_0)

print('input > output')
Z_1 = W_0@A_0
print(f'{Z_1 = }', end = '\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top input
[[ 1.  1.  1.]
 [ 0. -1.  1.]
 [-1.  0. -1.]]
input > output
Z_1 = array([[ 0.,  2., -1.],
       [-1., -1.,  0.],
       [-2., -3.,  0.]])

A_1 = array([[0., 2., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

1.1 2

[29]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 =}')

print('\ninput > output')
Z_1 = W_0@A_0
print(f'{Z_1 = }', end = '\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top input
A_0 =array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])

input > output
```

```
Z_1 = array([[ 0.,  2., -1.],
       [-1., -1.,  0.],
       [-2., -3.,  0.]])

A_1 = array([[0., 2., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[ ]:

exp1-ml

April 22, 2025

[11]:
```
import numpy as np
```

[16]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([1, -1])
theta = 1
Z = W@X
h1 = [int(i>=theta) for i in Z]
print(h1)
```

```
[0, 0, 1, 0]
```

[17]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([-1, 1])
theta = 1
Z = W@X
h2 = [int(i>=theta) for i in Z]
print(h2)
```

```
[0, 1, 0, 0]
```

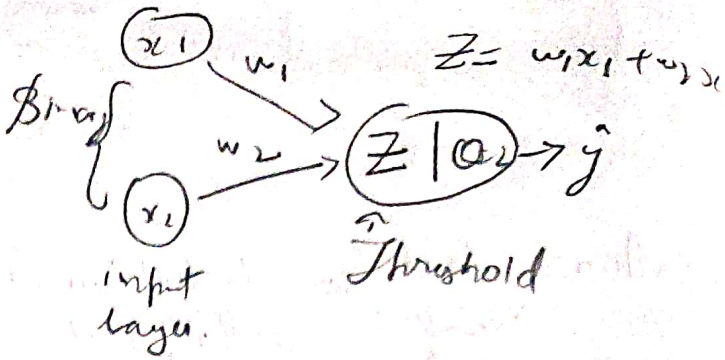
[18]:
```
X = np.array([h1, h2])

W = np.array([1, 1])
theta = 1
Z = W@X
XOR = [int(i>= theta) for i in Z]

print(XOR)
```

```
[0, 1, 1, 0]
```

[ ]:
```

```

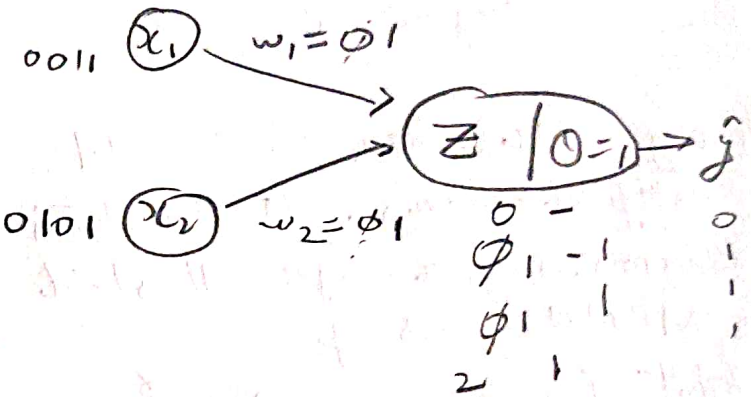
AND GATE

x_1	x_2	y	
0	0	0	False
0	1	0	False
1	0	0	False
1	1	1	True



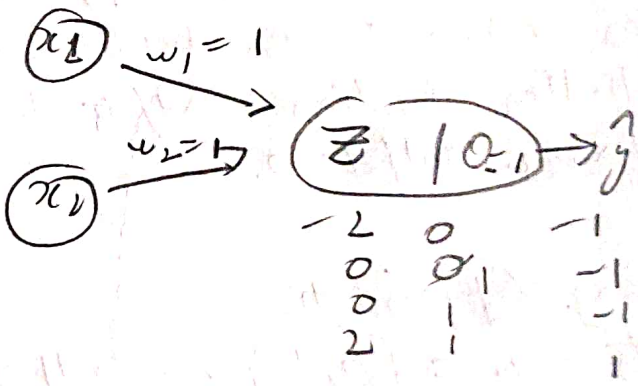
OR gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



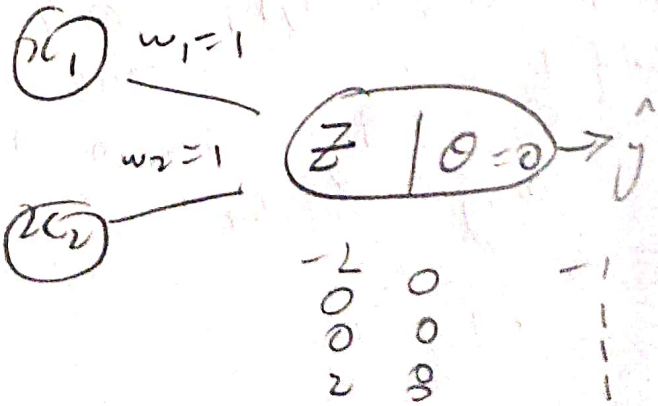
AND gate (Bifolar)

x_1	x_2	y
-1	-1	-1
-1	1	-1
1	-1	-1
1	1	1



OR gate (Bifolar)

x_1	x_2	y
-1	-1	-1
-1	1	1
1	-1	1
1	1	1



exp-1-ml

April 23, 2025

[2]:
```
import numpy as np
```

[3]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([1, 1])                    #Polar OR
theta = 1
Z = W@X
y_pred = [int(i>=theta) for i in Z]
print(y_pred)
```

```
[0, 1, 1, 1]
```

[4]:
```
X = np.array([[0, 0, 1, 1],
              [0, 1, 0, 1]])
W = np.array([1, 1])
theta = 2
Z = W@X                                 #Polar AND
y_pred = [int(i>=theta) for i in Z]
print(y_pred)
```

```
[0, 0, 0, 1]
```

[5]:
```
X = np.array([[-1, -1, 1, 1],
              [-1, 1, -1, 1]])
W = np.array([1, 1])
theta = 0                               #Bipolar OR
Z = W@X
y_pred = [int(i>=theta)*2-1 for i in Z]
print(y_pred)
```

```
[-1, 1, 1, 1]
```

[6]:
```
X = np.array([[-1, -1, 1, 1],
              [-1, 1, -1, 1]])
W = np.array([1, 1])                    #Bipolar AND
theta = 1
Z = W@X
y_pred = [int(i>=theta)*2-1 for i in Z]
```

exp1-ml

April 22, 2025

```
]: import numpy as np
```

```
]: X = np.array([[0, 0, 1, 1],
                 [0, 1, 0, 1]])
   W = np.array([1, -1])
   theta = 1
   Z = W@X
   h1 = [int(i>=theta) for i in Z]
   print(h1)
```

```
[0, 0, 1, 0]
```

```
]: X = np.array([[0, 0, 1, 1],
                 [0, 1, 0, 1]])
   W = np.array([-1, 1])
   theta = 1
   Z = W@X
   h2 = [int(i>=theta) for i in Z]
   print(h2)
```

```
[0, 1, 0, 0]
```

```
]: X = np.array([h1, h2])

   W = np.array([1, 1])
   theta = 1
   Z = W@X
   XOR = [int(i>= theta) for i in Z]

   print(XOR)
```

```
[0, 1, 1, 0]
```

```
]:
```

x_0	x_1	x_2	y	w_0	w_1	w_2	z	$\hat{y}$	$(y-\hat{y})$	effect
1	0	0	0	0	0	0	0	1	-1	w-x
1	0	1	0	-1	0	0	-1	0	1	w+x
1	1	0	1	0	0	1	0	1	0	NC
1	1	1	1	0	0	1	1	1	0	NC
1	0	0	0	0	0	1	0	1	-1	w-x
1	0	1	1	-1	0	1	-1	0	1	w+x
1	1	0	1	-1	0	1	0	1	0	NC
1	1	1	1	0	1	1	2	1	0	NC
1	0	0	0	0	1	1	0	1	-1	w-x
1	0	1	1	-1	1	1	-1	1	0	NC
1	1	0	1	-1	1	1	-1	1	0	NC
1	1	1	1	-1	1	1	-1	1	0	NC
1	0	0	0	-1	1	1	-1	0	0	NC
1	0	1	1	-1	1	1	-1	1	0	NC
1	1	0	1	-1	1	1	0	1	0	NC
1	1	1	1	-1	1	1	1	1	0	NC

Conclusion :-
The perceptron learning algorithm successfully learns to classify linearly separable data such as AND, OR, NOT. It updates the weights on misclassified samples.

April 22, 2025

```
5]: import numpy as np
```

0.0.1 Input

```
3]: X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
    print(f'{X.shape=}')
    X
```

```
X.shape=(2, 4)
```

```
3]: array([[0, 0, 1, 1],
           [0, 1, 0, 1]])
```

```
9]: t = np.array([[0, 0, 0, 1]])
    print(f'{t.shape=}')
    t
```

```
t.shape=(1, 4)
```

```
1]: array([[0, 1, 1, 1]])
```

0.0.2 Add bias

```
]: A = np.vstack((np.ones((1, 4)), X))
   A
```

```
]: array([[1., 1., 1., 1.],
          [0., 0., 1., 1.],
          [0., 1., 0., 1.]])
```

0.0.3 Initialize weights

```
]: W = np.array([[0, 0, 0]])
   print(f'{W.shape=}')
```

```
W.shape=(1, 3)
```

```
MAX_EPOCH = 10
```

0.0.4 Perceptron learning

```
for ep in range(MAX_EPOCH):
    print('Epoch-', ep+1, '='*50)
    isConv = True
    for x, y in zip(A.T, t.ravel()):
        Z = W @ x.T
        # Actually from positive but predicted as negative
        if y>0 and Z < 0:
            W = W + x
            isConv = False
            print('Update: W+x')
        # Actually from negative but predicted as positive
        if y<1 and Z >= 0:
            W = W - x
            isConv = False
            print('Update: W-x')
        print(x, y, Z, W)
    if isConv:
        print('Final weights: ', W)
        break
```

```
Epoch- 1 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  0.  0.]]
Update: W+x
[1. 0. 1.] 1 [-1.] [[0. 0. 1.]]
[1. 1. 0.] 1 [0.] [[0. 0. 1.]]
[1. 1. 1.] 1 [1.] [[0. 0. 1.]]
Epoch- 2 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  0.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  0.  1.]]
Update: W+x
[1. 1. 0.] 1 [-1.] [[0. 1. 1.]]
[1. 1. 1.] 1 [2.] [[0. 1. 1.]]
Epoch- 3 ==================================================
Update: W-x
[1. 0. 0.] 0 [0.] [[-1.  1.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 0.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 1.] 1 [1.] [[-1.  1.  1.]]
Epoch- 4 ==================================================
[1. 0. 0.] 0 [-1.] [[-1.  1.  1.]]
[1. 0. 1.] 1 [0.] [[-1.  1.  1.]]
```

```
[1. 1. 0.] 1 [0.] [[-1.  1.  1.]]
[1. 1. 1.] 1 [1.] [[-1.  1.  1.]]
Final weights:  [[-1.  1.  1.]]
```

]:

7]:
```
Z = W @ A
y_pred = [int(pred >= 0) for pred in Z.ravel()]
print(f'{y_pred=}')
```

```
y_pred=[0, 1, 1, 1]
```

]:

$$W^{[0]}_{(2\times3)} = \begin{bmatrix} -1 & -1 & 1 \\ -1 & 2 & 1 \end{bmatrix}$$

$$X_{(2\times3)} = \begin{bmatrix} 0 & -1 & 1 \\ -1 & 0 & -1 \end{bmatrix}$$

$f^{[1]}(\cdot)$ = "W - Sigmoid"

Layer 1, input 3, output = 2, examples = 3

$$Z^{[1]}_{(2\times3)} = W^{[0]}_{(2\times3)} \cdot A^{[0]}_{(3\times3)}$$

$$\begin{bmatrix} -2 & 0 & -3 \\ -2 & 0 & -3 \end{bmatrix} = \begin{bmatrix} -1 & -1 & 1 \\ -1 & 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 0 & -1 & 1 \\ -1 & 0 & -1 \end{bmatrix} \rightarrow \text{Big}$$

$$A^{[1]}_{(2\times3)} = f^{[1]}\left(Z^{[1]}_{2\times3}\right)$$

$$A^{[1]} = \begin{bmatrix} 0.1192, & 0.5, & 0.0474 \\ 0.1192, & 0.0474, & 0.5 \end{bmatrix}$$

Multilayer - Neural Network :-

$$W^{[0]} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \quad (2\times2)$$

↓ output of hidden → input n/w

$$W^{[1]} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \quad (3\times3)$$

↓ output of n/w → input nodes of hidden layer.

$$X = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}_{(2\times 3)}$$

(2 → variables, 3 → examples.)

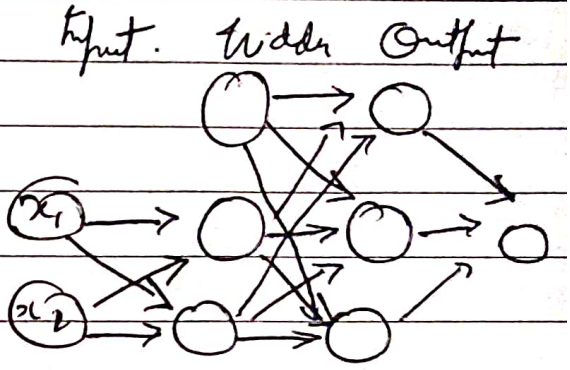
$f^{[1]}(\cdot)$ = linear

$f^{[2]}(\cdot)$ = Relu

1) Input → Hidden

$Z^{[1]} = W^{[0]} A^{[0]}$

$(2\times 3) \quad (2\times 2) \quad (2\times 3)$



$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

$$A^{[1]} = f^{[1]}\left(Z^{[1]}\right) = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

2) Hidden → Output

$Z^{[2]} = W^{[1]} \cdot A^{[1]}$

$(2\times 3) \cdot (3\times 3) \quad (3\times 3)$

$$\begin{bmatrix} 0 & -1 & -2 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

$A^{[2]} = f^{[2]}\left(Z^{[2]}\right)$

$$A^{[2]} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix} \quad \because f^{[2]}(\cdot) = \text{Relu}$$

Conclusion: →

The forward pass in a single layer perceptron calculates based on direct mapping of input, weights and biases, suitable for linearly

mlp-forward-pass-1-layered

April 22, 2025

3]:
```
import numpy as np
np.set_printoptions(precision=4)
```

4]:
```
W_0 = np.array([[-1, -1, 1],
                [-1, 2, 1]], dtype=float)
print(f'{W_0.shape=}')
```

```
W_0.shape=(2, 3)
```

5]:
```
X = np.array([[0, -1, 1],
              [-1, 0, -1]], dtype=float)
print(f'{X.shape=}')
```

```
X.shape=(2, 3)
```

6]:
```
f_1 = 'USigmoid'
```

7]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))

def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn='Lin'):
    A = np.array(list(map(globals()[fcn], Z)))
    return A
```

0.0.1 Forward pass

8]:
```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Output')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')
```

```
Pad bias at top of input
A_0 = array([[ 1.,  1.,  1.],
       [ 0., -1.,  1.],
       [-1.,  0., -1.]])



Input -> Output
Z_1 = array([[-2.,  0., -3.],
       [-2., -3.,  0.]])

A_1 = array([[0.1192, 0.5   , 0.0474],
       [0.1192, 0.0474, 0.5   ]])
```

]:

mlp-forward-pass-2-layered

April 22, 2025

8]:
```
import numpy as np
np.set_printoptions(precision=4)
```

9]:
```
W_0 = np.array([[1, -1],
                [1, -1]], dtype=float)
print(f'{W_0.shape=}')
```

```
W_0.shape=(2, 2)
```

0]:
```
W_1 = np.array([[-1, 1, 0,],
                [0, -1, 1],
                [1, 0, -1]], dtype=float)
print(f'{W_1.shape=}')
```

```
W_1.shape=(3, 3)
```

1]:
```
X = np.array([[1, 1, 0],
              [0, 1, 1]], dtype=float)
print(f'{X.shape=}')
```

```
X.shape=(2, 3)
```

12]:
```
f_1 = 'Lin'
f_2 = 'ReLU'
```

13]:
```
def USigmoid(x):
    return 1/(1 + np.exp(-x))

def BSigmoid(x):
    return (1 - np.exp(-x))/(1 + np.exp(-x))

def ReLU(x):
    return np.array([max(i, 0) for i in x])

def Lin(x):
    return x

def activation(Z, fcn='Lin'):
```

```
    A = np.array(list(map(globals()[fcn], Z)))
    return A
```

0.0.1 Forward pass

```
if X.shape[0] == W_0.shape[1]:
    print('No bias')
    A_0 = X
else:
    print('Pad bias at top of input')
    A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(f'{A_0 = }')

print('\n\nInput -> Hidden')
Z_1 = W_0 @ A_0
print(f'{Z_1 = }', end='\n\n')
A_1 = activation(Z_1, f_1)
print(f'{A_1 = }')

print('\n\nHidden -> Output')
if A_1.shape[0] == W_1.shape[1]:
    print('No bias')
else:
    print('Pad bias at top of A1')
    A_1 = np.vstack((np.ones((1, A_1.shape[1])), A_1))
print(f'{A_1 = }')

Z_2 = W_1 @ A_1
print(f'{Z_2 = }', end='\n\n')
A_2 = activation(Z_2, f_2)
print(f'{A_2 = }')
```

```
No bias
A_0 = array([[1., 1., 0.],
       [0., 1., 1.]])


Input -> Hidden
Z_1 = array([[ 1.,  0., -1.],
       [ 1.,  0., -1.]])

A_1 = array([[ 1.,  0., -1.],
       [ 1.,  0., -1.]])


Hidden -> Output
Pad bias at top of A1
```

Aim - To train single layer perceptron (Stochastic gradient descent).

Theory :→

SGD is an optimization algorithm used to minimize the error by updating the weights iteratively for each training sample. Loss function MSE is commonly used.

Perceptron Training Rule :

i) Forward Pass — Compute the weighted sum of output using activation function.

$$z = \sum_{i=1}^{n} w_i x_i, \quad y = \begin{cases} 1 & \text{if } s \geq 0 \\ 0 & s < 0 \end{cases}$$

ii) Error — Compute the error for each input $e = y - \hat{y}$

iii) Weight Update Rule : Adjust the weights and bias using learning rate.

iv) Repeat, For each epoch iterate over all training examples until, the error is minimized.

3) Procedure

$w^{[0]} = [1\ 0\ 1\ 1]$ (1×3) $\quad t = [1\ 1\ 1]$

$x = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 3 & 3 \end{bmatrix}$ (2×3) $\quad f^{[1]} = \text{sigmoid}$, $\alpha = 1$

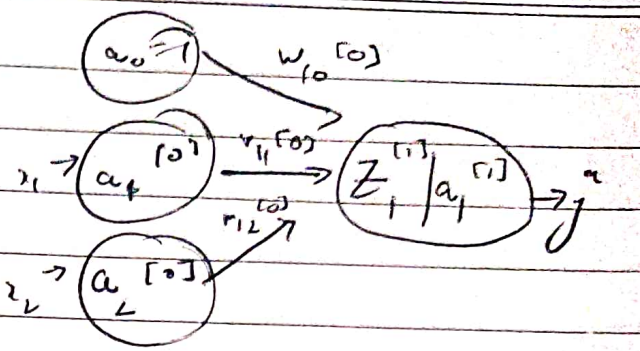
(1) Iteration - 1

$x = [1\ 3\ 0]^T$, $t = [1]$

Forward Pass

$$A^{[0]} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}, \quad Z_1^{[1]} = w^{[0]} A^{[0]} \quad [1\ 1\ 1] \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} = [5]$$

$A^{[2]} = f^{[1]T} z^{[1]} = [0.993]$



Error :- $\frac{1}{2}(0.9993 - 1)^2 \neq 0$

Apply Backward pass :-

$\Delta w^{[0]} = (a_1^{[1]} - \epsilon) \cdot f^{[1]}(\cdot)' * A[0]^T$

$= (0.0993 - 1) * [0.00663] \cdot [1 \quad 1 \quad 3]$

$= [-4.449e-05, \quad -4.49e-05, \quad -1.334e-04]$

$w_{new} = [1 \quad 1 \quad 1] - 1[-4.449e-05, \quad -4.49e-05, \quad -1.334e-04]$

$= [1, \quad 1, \quad 1.0001]$

Iteration 2

$w^T x = [1 \quad 3], \quad t = 1$

Forward Pass

$A^{[0]} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}, \quad z_1^{[1]} = w^{[0]} A^{[0]} =$

$[1 \quad 1 \quad 1] \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} = [5]$

$\hat{y} = A^{[1]} = f^{[1]} z^{[1]} = 0.9933$

Error $= \frac{1}{2}(0.9933 - 1)^2 \neq 0$

Apply Backward Pass =>

$\Delta w^{[0]} = (a_1^{(1)} - \epsilon) \cdot f^{[1]}(\cdot)' * A^{[0]T} =$

$= [-4.449e-05, -4.49e-05, -1.334e-04]$

$w_{new} = w_{old} - \alpha [\nabla w^{[0]}]$

$= [1, 1, 1.00003]$

Iteration 3

$x = [-1 \quad 2 \quad 3]^T$, $t = [1]$

$A^{[0]} = \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix}$; $Z_1^{(1)} = [1 \quad 1 \quad 1] \begin{bmatrix} 1 \\ -1 \\ 3 \end{bmatrix} = 0.9930$

$y = A^{[1]} = f^{(1)} z^{(1)} = 0.9930$

$error = \frac{1}{2} (0.9930 - 1)^2 = 0$

Apply Backward Pass ⇒

$w^{[0]} = [(0.9930 - 1) \quad (0.0177) \quad [1 \quad -1 \quad 3]]$

$= [-4.4408e-05, \quad -4.4408e-05, \quad -1.3323e-04]$

$dE/dw = 0$

$w_{new} = w_{old} - \alpha w^{[0]} = [1.0001, 1.0001, 1.0004]$

mlp-backward-pass-1-layered

April 22, 2025

0]:
```
import numpy as np
np.set_printoptions(precision=4)
```

1]:
```
W_0 = np.array([[1, 1, 1],], dtype=float)
```

2]:
```
X = np.array([[1, 1, 1],
              [3, 3, 3]], dtype=float)
```

3]:
```
t = np.array([[1, 1, 1],], dtype=float)
```

4]:
```
f_1 = 'USigmoid'
```

5]:
```
lr = 1
```

6]:
```
def USigmoid(x, direction):
    if direction == 'F':
        return 1/(1 + np.exp(-x))
    else:
        return USigmoid(x, 'F')*(1-USigmoid(x, 'F'))

def BSigmoid(x, direction):
    if direction == 'F':
        return (1 - np.exp(-x))/(1 + np.exp(-x))
    else:
        return 0.5*(1-BSigmoid(x, 'F')**2)

def ReLU(x, direction):
    if direction == 'F':
        return max(x, 0)
    else:
        return float(x > 0)

def Lin(x, direction):
    if direction == 'F':
        return x
    else:
        return 1
```

```
def activation(Z, fcn='Lin', direction = 'F'):
    A = np.array(list(map(globals()[fcn], Z, np.repeat(direction, Z.shape[0]))))
    return A
```

[17]:
```
MAX_EPOCHS = 1
```

[18]:
```
print('Pad bias at top of input')
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
print(A_0)
for ep in range(MAX_EPOCHS):
    print('\nEPOCH-', ep+1, '='*80)
    for itr, (x, y) in enumerate(zip(A_0.T, t.T)):
        print('\nITER-', itr+1, '-'*80)
        ### Forward pass
        print('Input -> Output')
        Z_1 = W_0 @ x
        print(f'{Z_1 = }')
        A_1 = activation(Z_1, f_1)
        print(f'{A_1 = }')

        ### Backward pass
        print('\nOutput -> Input')
        Error = 0.5*(A_1 - t)**2
        print(f'{Error = }')
        dE_dw = (A_1 - y) * activation(Z_1, f_1, 'B') * x
        print(f'{dE_dw = }')
        W_0 = W_0 - lr*dE_dw
        print(f'{W_0 = }')
```

```
Pad bias at top of input
[[1. 1. 1.]
 [1. 1. 1.]
 [3. 3. 3.]]

EPOCH- 1
================================================================================

ITER- 1
--------------------------------------------------------------------------------
Input -> Output
Z_1 = array([5.])
A_1 = array([0.9933])

Output -> Input
Error = array([[2.2397e-05, 2.2397e-05, 2.2397e-05]])
dE_dw = array([-4.4494e-05, -4.4494e-05, -1.3348e-04])
```

Aim - To train multilayer perceptron (stochastic gradient layer)

Libraries.

Theory :→

An MLP consists of an input layer, one or more hidden layers, and an output layer.

Forward Pass-

1) Input layer, 2) Hidden layer :-

Compute the weighted sum z and apply to activation function.

3) Output layer :-

similar computation like Hidden layer to get the output.

Backward Pass :-

1. Compute error :-

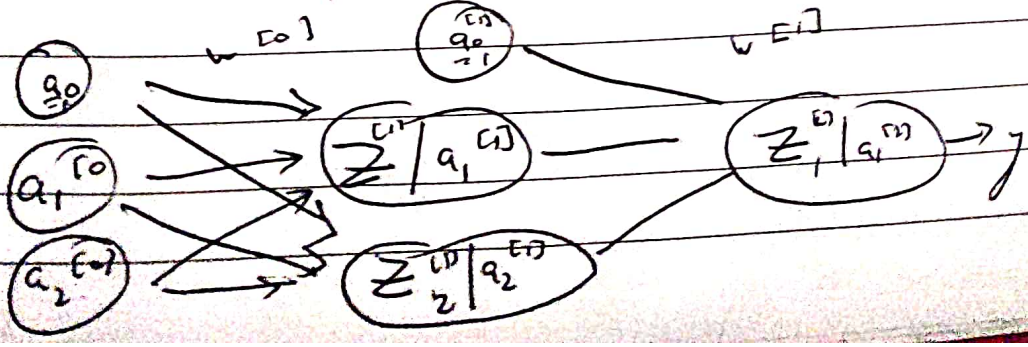
2. Back propagation, Output layer → Hidden layer.

Numerical :-

$$W^{[0]} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & 1 \end{bmatrix}_{2\times 3} \qquad W^{[1]} = [0 \; 1 \; -1]_{1\times 3}$$

$$x = \begin{bmatrix} 1 & -1 & 1 \\ 0 & -1 & 1 \end{bmatrix}_{4\times 3}, \quad t = [0 \; 0 \; 1]_{1\times 3}$$

$f^{[1]}$ = linear, $f^{[2]}$ = ReLU, Max Ep = 1



Using Stoch G.D

1) Epoch-1

Word-1 $x = [1\ 0]^T$, $t = 0$

Forward: (i) Input → Hidden (Layer 0 → Layer 1)

$$A^{[0]} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad Z^{[1]} = W^{[0]} A^{[0]}$$

$$\begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

$$A^{[1]} = f^{(1)} Z^{(1)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

(ii) Hidden → output (Layer 1 – Layer 2)

$$A^{[1]} = \begin{bmatrix} 1 \\ A[1] \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix}$$

$$Z^{(2)} = W^{[1]} A^{[1]} = \begin{bmatrix} 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ -2 \end{bmatrix} = 3$$

$$A^{(3)} = f^{[2]}\left(Z^{[2]}\right) = 3$$

Error MSE = $E = \frac{1}{2}(a_1^{[2]} - t)^2 = \frac{1}{2}(3-0)^2 = 4.5$

Backward: Output → Hidden

$$\Delta W_E^{[1]} = \left(A^{[2]} - t\right) \cdot f\left(Z^{[2]}\right) \cdot A^{[1]T}$$ with big

$$= (3-0) \cdot 1 \begin{bmatrix} 1 & 1 & -2 \end{bmatrix} = \begin{bmatrix} 3 & 3 & -6 \end{bmatrix}$$

ii) Hidden - Input

$$\Delta w^{[0]} = \left(A^{[2]} - t\right) f'^{[2]}(z^{[2]})$$

(2×3) $\quad 1\times 1$ $\quad 1\times 1$

$$f^{[1]}(z^{[1]}) \times w^{[1]T}$$ without bias $\quad \Delta^{[0]T}$ with bias

$$= (3-0)\cdot 1\left(\begin{bmatrix}1\\1\end{bmatrix}\times\begin{bmatrix}1\\1\end{bmatrix}\right) \quad [1\ 1\ 0]$$

$$\Delta w^{[0]} = 3\begin{bmatrix}1\\-1\end{bmatrix}\cdot[1\ 1\ 0] = \begin{bmatrix}3 & 3 & 0\\ 3 & 3 & 0\end{bmatrix}$$

weight update

$$w_{new}^{[1]} := w_{old}^{[1]} - \alpha\,\Delta w^{[1]}$$

Assume $\alpha = 1$

$$w^{[1]} = [0\ 1\ -1] - [3\ 3\ -3] = [-3\ -2\ 2]$$

$$w^{[0]} = \begin{bmatrix}1 & 0 & 1\\ -1 & -1 & 1\end{bmatrix} - \begin{bmatrix}3 & 3 & 0\\ -3 & -3 & 0\end{bmatrix} = \begin{bmatrix}-2 & -3 & 1\\ 2 & 2 & 1\end{bmatrix}$$

Conclusion

MLP backward pass computes gradient via backpropagation enabling efficient weight updates for complex input output relationships

mlp-backward-pass-2-layered-1

April 23, 2025

[7]:
```
import numpy as np
np.set_printoptions(precision=4)

# Initialize weights
W_0 = np.array([[1, 0, 1],
                [-1, -1, 1]], dtype=float)  # Hidden layer weights
W_1 = np.array([[0, 1, -1]], dtype=float)   # Output layer weights

# Input and target
X = np.array([[1, -1, 1],
              [0, -1, 1]], dtype=float)     # Shape: (2, 2)
t = np.array([[0, 0, 1]], dtype=float)   # Shape: (1, 2)

# Hyperparameters
f_1 = "Lin"
f_2 = "ReLU"
lr = 1
MAX_EPOCHS = 1

# Activation Functions
def USigmoid(x, direction='F'):
    fx = 1 / (1 + np.exp(-x))
    return fx if direction == 'F' else fx * (1 - fx)

def BSigmoid(x, direction='F'):
    fx = (1 - np.exp(-x)) / (1 + np.exp(-x))
    return fx if direction == 'F' else 0.5 * (1 - fx ** 2)

def TanH(x, direction='F'):
    fx = np.tanh(x)
    return fx if direction == 'F' else 1 - fx ** 2

def ReLU(x, direction='F'):
    return np.maximum(0, x) if direction == 'F' else (x > 0).astype(float)

def Lin(x, direction='F'):
    return x if direction == 'F' else np.ones_like(x)
```

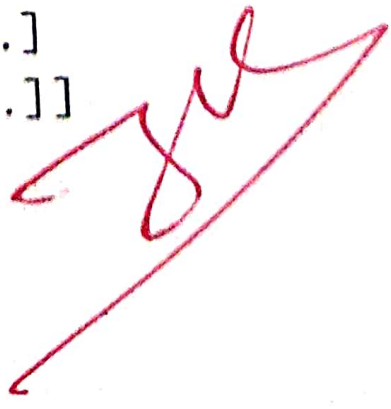
```
# Wrapper for activation
def activation(Z, fcn="Lin", direction='F'):
    return np.array([globals()[fcn](z, direction) for z in Z]).reshape(-1, 1)

A_0 = np.vstack((np.ones((1, 1)), x.reshape(-1, 1)))

# Training loop
for ep in range(MAX_EPOCHS):
    print('\nEPOCH-', ep + 1, '=' * 80)
    for itr, (x, y) in enumerate(zip(X.T, t.T)):
        print(f'\nITER-{itr + 1} ' + '-' * 80)

        # Forward Pass: Input -> Hidden
        A_0 = x.reshape(-1, 1)

        # Conditionally add bias to A_0
        if W_0.shape[1] == A_0.shape[0] + 1:
            A_0 = np.vstack((np.ones((1, 1)), A_0))  # Add bias at the top
            print('Bias added to A_0')

        Z_1 = W_0 @ A_0
        A_1 = activation(Z_1, f_1)

        # Forward Pass: Hidden -> Output
        # Conditionally add bias to A_1
        if W_1.shape[1] == A_1.shape[0] + 1:
            A_1 = np.vstack((np.ones((1, 1)), A_1))  # Add bias at the top
            print('Bias added to A_1')

        Z_2 = W_1 @ A_1
        A_2 = activation(Z_2, f_2)

        print(f'A_0 (input):\n{A_0}')
        print(f'Z_1 (hidden pre-activation):\n{Z_1}')
        print(f'A_1 (hidden post-activation):\n{A_1}')
        print(f'Z_2 (output pre-activation):\n{Z_2}')
        print(f'A_2 (output post-activation):\n{A_2}')

       # Backward Pass: Output -> Hidden
        delta_2 = (A_2 - y.reshape(-1, 1)) * activation(Z_2, f_2, 'B')
        dW_1 = delta_2 @ A_1.T

        # Remove bias from W_1 if necessary
        if W_1.shape[1] == Z_1.shape[0] + 1:
            W_1_no_bias = W_1[:, 1:]
            print("Bias column removed from W_1 for backpropagation.")
```

```
        else:
            W_1_no_bias = W_1

        print(W_1_no_bias)

        # Backward Pass: Hidden -> Input
        delta_1 = (W_1_no_bias.T @ delta_2) * activation(Z_1, f_1, 'B')
        dW_0 = delta_1 @ A_0.T

        # Update weights
        W_1 -= lr * dW_1
        W_0 -= lr * dW_0

        print(f'dW_1:\n{dW_1}')
        print(f'dW_0:\n{dW_0}')
        print(f'Updated W_1:\n{W_1}')
        print(f'Updated W_0:\n{W_0}')
```

```
EPOCH- 1
================================================================================

ITER-1
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[1.]
 [1.]
 [0.]]
Z_1 (hidden pre-activation):
[[ 1.]
 [-2.]]
A_1 (hidden post-activation):
[[ 1.]
 [ 1.]
 [-2.]]
Z_2 (output pre-activation):
[[3.]]
A_2 (output post-activation):
[[3.]]
Bias column removed from W_1 for backpropagation.
[[ 1. -1.]]
dW_1:
[[ 3.  3. -6.]]
dW_0:
[[ 3.  3.  0.]
```

```
 [-3. -3.  0.]]
Updated W_1:
[[-3. -2.  5.]]
Updated W_0:
[[-2. -3.  1.]
 [ 2.  2.  1.]]

ITER-2
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[ 1.]
 [-1.]
 [-1.]]
Z_1 (hidden pre-activation):
[[ 0.]
 [-1.]]
A_1 (hidden post-activation):
[[ 1.]
 [ 0.]
 [-1.]]
Z_2 (output pre-activation):
[[-8.]]
A_2 (output post-activation):
[[0.]]
Bias column removed from W_1 for backpropagation.
[[-2.  5.]]
dW_1:
[[0. 0. 0.]]
dW_0:
[[0. 0. 0.]
 [0. 0. 0.]]
Updated W_1:
[[-3. -2.  5.]]
Updated W_0:
[[-2. -3.  1.]
 [ 2.  2.  1.]]

ITER-3
--------------------------------------------------------------------------------
Bias added to A_0
Bias added to A_1
A_0 (input):
[[1.]
 [1.]
 [1.]]
Z_1 (hidden pre-activation):
```

```
[[-4.]
 [ 5.]]
A_1 (hidden post-activation):
[[ 1.]
 [-4.]
 [ 5.]]
Z_2 (output pre-activation):
[[30.]]
A_2 (output post-activation):
[[30.]]
Bias column removed from W_1 for backpropagation.
[[-2.  5.]]
dW_1:
[[  29. -116.  145.]]
dW_0:
[[-58. -58. -58.]
 [145. 145. 145.]]
Updated W_1:
[[ -32.  114. -140.]]
Updated W_0:
[[  56.   55.   59.]
 [-143. -143. -144.]]
```



Aim:- To study the tensorflow framework for graph calculation.

Theory:-
Tensorflow is an open source machine learning framework developed by Google. It allows for efficient computation of mathematical operations and is particularly used for building, training and deploying deep learning models.

Implementation:-

$W_0 = [0.1, -0.2, 0.1]$

$b = [0.1, -0.1, -0.1]$

$X = \begin{bmatrix} 1 & 1 & -1 \\ 0 & 1 & 1 \end{bmatrix}$

$f^{[1]} = f_1$, $\alpha = 0.1$

$Z_1^{[1]} = W^{[0]} \cdot A^{[0]}$

$Z^{[1]}_{(3\times 0)} = [0.1 \quad -0.2 \quad 0.1] \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = (1)$

$A^{[1]} = 1$

error = 0, No Backward pass...

$W \Rightarrow$ [Iteration: -> 2

$Z_1^{[1]} = W^{[0]} \cdot A^{[0]} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \times [1 \; 0 \; 1] = [2]$

$A^{[1]} = \text{sig}(A^{[0]}) = [2] = 2$

No error.

Conclusion:- Tensorflow efficiently tape tracks operations and automatic differentiation

```
c:\users\admin\.conda\envs\sclab\lib\site-packages (from matplotlib)
(2.9.0.post0)
Requirement already satisfied: six>=1.5 in
c:\users\admin\.conda\envs\sclab\lib\site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)
Downloading matplotlib-3.10.0-cp311-cp311-win_amd64.whl (8.0 MB)
   ---------------------------------------- 0.0/8.0 MB ? eta -:--:--
   ---------- ----------------------------- 3.1/8.0 MB 20.5 MB/s eta 0:00:01
   ---------------------------------------- 8.0/8.0 MB 21.6 MB/s eta 0:00:00
Downloading contourpy-1.3.1-cp311-cp311-win_amd64.whl (219 kB)
Using cached cycler-0.12.1-py3-none-any.whl (8.3 kB)
Downloading fonttools-4.56.0-cp311-cp311-win_amd64.whl (2.2 MB)
   ---------------------------------------- 0.0/2.2 MB ? eta -:--:--
   ---------------------------------------- 2.2/2.2 MB 62.5 MB/s eta 0:00:00
Downloading kiwisolver-1.4.8-cp311-cp311-win_amd64.whl (71 kB)
Downloading pillow-11.1.0-cp311-cp311-win_amd64.whl (2.6 MB)
   ---------------------------------------- 0.0/2.6 MB ? eta -:--:--
   ---------------------------------------- 2.6/2.6 MB 50.3 MB/s eta 0:00:00
Downloading pyparsing-3.2.1-py3-none-any.whl (107 kB)
Installing collected packages: pyparsing, pillow, kiwisolver, fonttools, cycler,
contourpy, matplotlib
Successfully installed contourpy-1.3.1 cycler-0.12.1 fonttools-4.56.0
kiwisolver-1.4.8 matplotlib-3.10.0 pillow-11.1.0 pyparsing-3.2.1
```

Load library

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

[21]:
```
a = tf.constant(3e-38)
type(a), a.shape, a.dtype
```

[21]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([]), tf.float32)
```

[7]:
```
a = tf.constant([3, 2, 3.])
type(a), a.shape, a.dtype
```

[7]:
```
(tensorflow.python.framework.ops.EagerTensor, TensorShape([3]), tf.float32)
```

[22]:
```
a = tf.Variable([[3, 2., 3], [3, 2, 3]])
type(a), a.shape, a.dtype
```

[22]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 3]),
 tf.float32)
```

[26]:
```
a = tf.Variable([[[[[3, 2, 3], [3, 2, 3]],]]])
type(a), a.shape, a.dtype
```

[26]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([1, 1, 1, 2, 3]),
 tf.int32)
```

[13]:
```
a = tf.Variable([[[3, 2, 3], [3, 2, 3]],[[3, 2, 3], [3, 2, 3]]])
type(a), a.shape, a.dtype
```

[13]:
```
(tensorflow.python.ops.resource_variable_ops.ResourceVariable,
 TensorShape([2, 2, 3]),
 tf.int32)
```

Gradient tape

[34]:
```
a = tf.constant([2.])
x = tf.Variable([3.,4])
y = 2*x + a
```

[35]:
```
y.shape, y.dtype, y.numpy()
```

[35]:
```
(TensorShape([2]), tf.float32, array([ 8., 10.], dtype=float32))
```

[41]:
```
with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = 2*x**1 + a
```

[42]:
```
dy_dx = tape.gradient(y, x).numpy()
dy_dx
```

[42]:
```
array([2., 2.], dtype=float32)
```

[43]:
```
a = tf.constant([2.])
w1 = tf.Variable([3.,])
w2 = tf.Variable([1.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(w1)
    tape.watch(w2)

    y = 2*w1**2 + w2**3 + a
```

[45]:
```
dy_dw1, dy_dw2 = tape.gradient(y, [w1, w2])
dy_dw1.numpy(), dy_dw2.numpy()
```

[45]:
```
(array([12.], dtype=float32), array([3.], dtype=float32))
```

[46]:
```
a = tf.constant([2.])
W = tf.Variable([3., 2.])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(W)

    y = 2*W[0]**2 + W[1]**3 + a
```

[48]:
```
tape.gradient(y, W)
```

[48]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([12., 12.], dtype=float32)>
```

Sigmoid derivative

[22]:
```
x = tf.Variable([3.,])

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)

    y = 1/(1 + tf.exp(-x))

dy_dx = tape.gradient(y, x)
```

[23]:
```
y, dy_dx
```

[23]:
```
(<tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.95257413],
dtype=float32)>,
 <tf.Tensor: shape=(1,), dtype=float32, numpy=array([0.04517666],
dtype=float32)>)
```

[49]:
```
x = tf.Variable(np.arange(-5, 5, 0.1))

with tf.GradientTape(persistent=True) as tape:
    tape.watch(x)
    y = 1/(1 + tf.exp(-x))

dy_dx = tape.gradient(y, x)
```

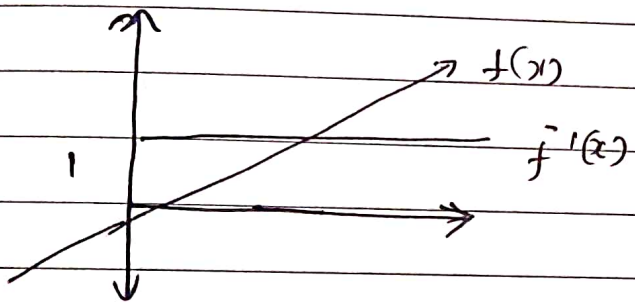
[50]:
```
plt.plot(x.numpy(), y.numpy(), 'b')
plt.plot(x.numpy(), dy_dx.numpy(), 'r')
plt.show()
```

Aim: To study different activation function and their derivatives using gradient tape.

Theory:-

Activation Function:-

(i) Linear simply returns the input value directly identity function. They are sometimes used in output layers for regression tasks where the output is not bounded to a specific range. $f(x) = x$, $f'(x) = 1$



(ii) Unipolar Sigmoid -

Squashes any input value to a range between 0 and 1. Used in output layer for binary classification.

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

$$f'(x) = -1(1+e^{-x})^{-2} \times e^{-x}$$

$$= e^{-x}(1+e^{-x})^{-2}$$

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

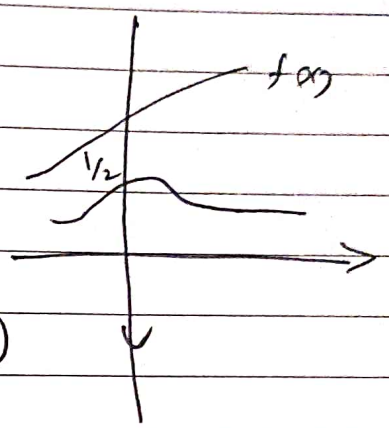
$$f'(x) = f(x) \cdot (1 - f(x))$$

Conclusion :- Thus we implemented CNN for binary classification. Balanced model simplifies and performance achieving accurate predictions.

iii) Bipolar sigmoid – similar to sigmoid but output values between 1 and −1. Can be used in hidden layers as it can handle negative inputs better than sigmoid

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

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



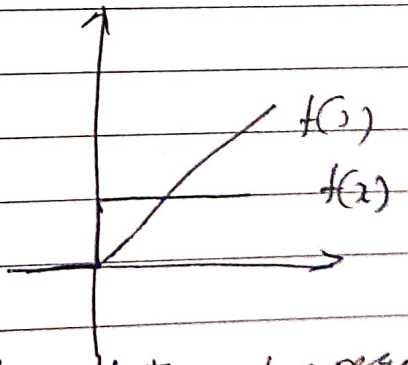
$$f(x)' = \frac{(1 + e^{-x})(e^{-x}) - (1 - e^{-x}).(e^{-x})}{(1 + e^{-x})^2.}$$

$$f'(x) = \frac{1}{2}(1 + f(x)^2)$$

Relu – If input is positive it output the input as directly. If the input is −ve, it outputs 0.

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0 & x < 0 \end{cases}$$

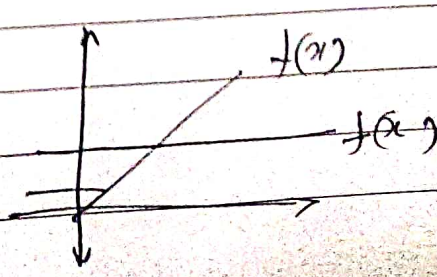
$$f'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



v) Leaky Relu : Similar to Relu instead of outputting for negative inputs it outputs, as a small fraction of the input (αx)

$$f(x) = \begin{cases} x & , x \geq 0 \\ \alpha x & x < 0 \end{cases}$$

$$f'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



```
W_0 = array([[ 0.43, -0.03,  0.43]])
```

[4]:
```
import numpy as np
import tensorflow as tf

# Initial weights
W_0 = tf.Variable([[1.0, 0.0, 1.0]], dtype=tf.float32)
print("Initial W_0 shape:", W_0.shape)

# Targets
t = tf.constant([[1.0, -1.0, -1.0]], dtype=tf.float32)

# Inputs
X = tf.constant([
    [1.0, 1.0, -1.0],
    [0.0, 1.0, 1.0]
], dtype=tf.float32)
print("X shape:", X.shape)

# Activation function name
f_1 = "Lin"

# Learning rate
lr = 0.1

# Activation Functions
def USigmoid(x):
    return tf.math.sigmoid(x)

def USigmoid_deriv(x):
    fx = tf.math.sigmoid(x)
    return fx * (1 - fx)

def BSigmoid(x):
    return (1 - tf.exp(-x)) / (1 + tf.exp(-x))

def BSigmoid_deriv(x):
    fx = BSigmoid(x)
    return 0.5 * (1 - tf.square(fx))

def ReLU(x):
    return tf.nn.relu(x)

def ReLU_deriv(x):
    return tf.cast(x > 0, tf.float32)

def Lin(x):
```

```
    return x

def Lin_deriv(x):
    return tf.ones_like(x)

# Activation function dictionary
activation_map = {
    "USigmoid": (USigmoid, USigmoid_deriv),
    "BSigmoid": (BSigmoid, BSigmoid_deriv),
    "ReLU": (ReLU, ReLU_deriv),
    "Lin": (Lin, Lin_deriv),
}

# Select activation function and its derivative
activation_fn, activation_deriv = activation_map[f_1]

# Add bias term to input (pad bias row at the top)
A_0 = np.vstack((np.ones((1, X.shape[1])), X))
A_0 = tf.constant(A_0, dtype=tf.float32)

# Training loop
MAX_EPOCH = 1

# Epoch loop
# Epoch loop
for ep in range(MAX_EPOCH):
    print(f"\nEPOCH-{ep + 1} " + "=" * 80)

    for itr, (x, y) in enumerate(zip(tf.transpose(A_0), tf.transpose(t))):
        print(f"\nITER-{itr + 1} " + "-" * 80)

        # Forward pass
        Z_1 = tf.matmul(W_0, tf.reshape(x, [-1, 1]))
        A_1 = activation_fn(Z_1)

        print(f"Target (y): {y.numpy()[0]:.2f}")
        print(f"Z_1 (Weighted sum): {Z_1.numpy()[0][0]:.4f}")
        print(f"A_1 (Activated output): {A_1.numpy()[0][0]:.4f}")

        # Error and gradient
        Error = 0.5 * (A_1 - y) ** 2
        dE_dZ = (A_1 - y) * activation_deriv(Z_1)
        dE_dW = tf.matmul(dE_dZ, tf.reshape(x, [1, -1]))

        print(f"Error: {Error.numpy()[0][0]:.4f}")
        print(f"dE_dW (Gradients): {np.round(dE_dW.numpy(), 4)}")
```

```
        # Update weights
        W_0.assign_sub(lr * dE_dW)
        print(f"Updated W_0: {np.round(W_0.numpy(), 4)}")

Initial W_0 shape: (1, 3)
X shape: (2, 3)

EPOCH-1
================================================================================

ITER-1
--------------------------------------------------------------------------------
Target (y): 1.00
Z_1 (Weighted sum): 1.0000
A_1 (Activated output): 1.0000
Error: 0.0000
dE_dW (Gradients): [[0. 0. 0.]]
Updated W_0: [[1. 0. 1.]]

ITER-2
--------------------------------------------------------------------------------
Target (y): -1.00
Z_1 (Weighted sum): 2.0000
A_1 (Activated output): 2.0000
Error: 4.5000
dE_dW (Gradients): [[3. 3. 3.]]
Updated W_0: [[ 0.7 -0.3  0.7]]

ITER-3
--------------------------------------------------------------------------------
Target (y): -1.00
Z_1 (Weighted sum): 1.7000
A_1 (Activated output): 1.7000
Error: 3.6450
dE_dW (Gradients): [[ 2.7 -2.7  2.7]]
Updated W_0: [[ 0.43 -0.03  0.43]]
```

components:

- **Input Layer:** Receives input images for classification.
- **Convolutional Layers:** Extract features from the images through convolutional operations.
- **Pooling Layers:** Reduce the spatial dimensions of the feature maps.
- **Flatten Layer:** Convert the 2D feature maps into a 1D vector.
- **Fully Connected Layers:** Perform classification using densely connected layers.
- **Output Layer:** Provides the final prediction probabilities for cat and dog classes.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
GlobalAveragePooling2D, Dense, Dropout, BatchNormalization

# Define constants
image_size = 128
image_channel = 3
num_classes = 2  # Use softmax for 2 classes instead of sigmoid
input_shape = (image_size, image_size, image_channel)

model = Sequential()
model.add(Input(shape=input_shape))  # ✅ Consistent Input layer

# Conv Block 1
model.add(Conv2D(40, kernel_size=(3,3), activation='relu',
```

```
padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 2
model.add(Conv2D(72, kernel_size=(3,3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 3
model.add(Conv2D(144, kernel_size=(3,3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 4
model.add(Conv2D(220, kernel_size=(3,3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Conv Block 5
model.add(Conv2D(280, kernel_size=(3,3), activation='relu',
padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

# Global Average Pooling + Dense
model.add(GlobalAveragePooling2D())
model.add(Dense(96, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(num_classes, activation='softmax'))  # □ Softmax for
2-class classification

model.summary()
```

```
Model: "sequential_7"
```

Layer (type)	Output Shape	Param #
conv2d_23 (Conv2D)		