Concept of Batch Normalization: Batch normalization is a technique used in artificial neural networks to stabilize and accelerate training by normalizing the inputs to each layer within a mini-batch. It reduces the internal covariate shift, which occurs when the distribution of activations changes during training, helping the network to converge faster.

Benefits of Using Batch Normalization During Training: Faster Convergence: Batch normalization reduces the sensitivity to the choice of hyperparameters, enabling faster training. Higher Learning Rates: With more stable gradients, it allows the use of higher learning rates, speeding up the optimization process. Reduces Overfitting: By adding a slight regularization effect, it minimizes overfitting, even without dropout. Improved Gradient Flow: Prevents gradients from becoming too large or small, reducing vanishing or exploding gradient problems. Better Initialization Robustness: The network becomes less sensitive to the initialization of weights, improving training stability. Working Principle of Batch Normalization: Batch normalization involves the following steps:

1. **Normalization: For a mini-batch of activations x**

   { x 1 , x 2 ,
2.
3.
4. , x m } x={x 1,x 2,...,x m} from a layer:

Compute the mean (μ µ) and variance (σ 2 σ 2) of the mini-batch: μ B

1 m ∑ i

1 m x i , σ B 2

1 m ∑ i

1 m (x i − μ B) 2 µ B

m 1

i=1 ∑ mx i,σ B 2

m 1

i=1 ∑ m(x i−μ B) 2

Normalize the activations: x ^ i

x i − μ B σ B 2 + ϵ x ^

i

σ B 2+ϵ

x i−μ B

Here, ϵ ϵ is a small constant added for numerical stability. 2. Scaling and Shifting: To allow the model to learn an optimal representation, two learnable parameters, γ γ (scale) and β β (shift), are introduced:

y i

γ x ^ i + β y i=γ x ^

i+β These parameters enable the network to restore any lost expressiveness from normalization.

3. Update During Training: During training, μ B μ Band σ B 2 σ B 2are computed for each mini-batch. During inference, a moving average of μ B μ Band σ B 2 σ B 2 (calculated during training) is used instead of mini-batch statistics to ensure consistent behavior.

In [2]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import os
```

In [4]:
```
tf.__version__
```

Out[4]: '2.18.0'

In [5]:
```
tf.config.list_physical_devices("GPU")
```

Out[5]: []

In [6]:
```
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
```

In [8]:
```
print(f"data type of x_train:{x_train.dtype},\n shape{x_train.shape}")
```

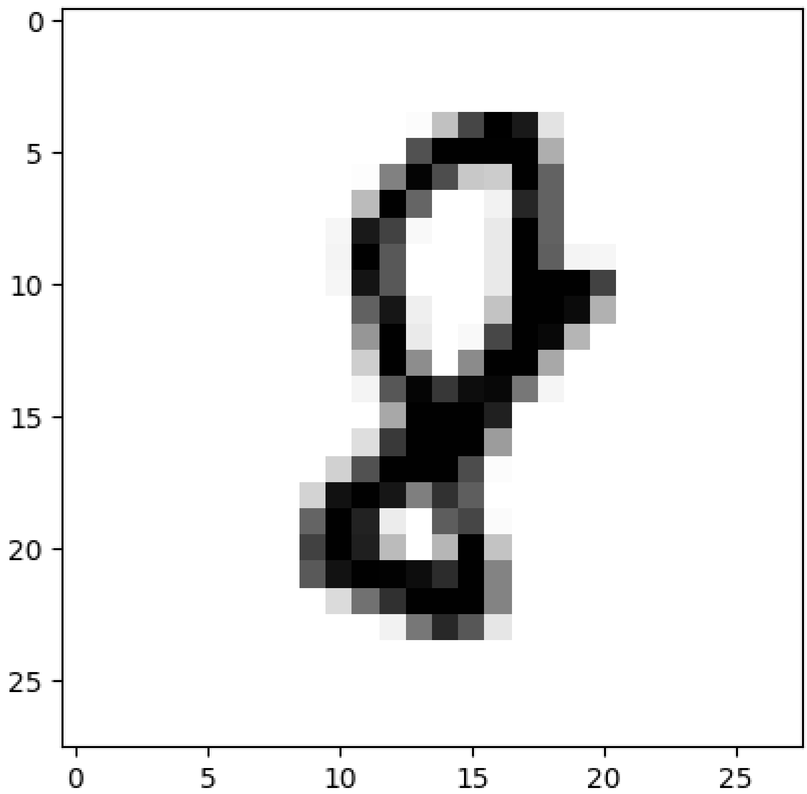
```
data type of x_train:uint8,
 shape(60000, 28, 28)
```

In [10]:
```
x_test.shape
```

Out[10]:
```
(10000, 28, 28)
```

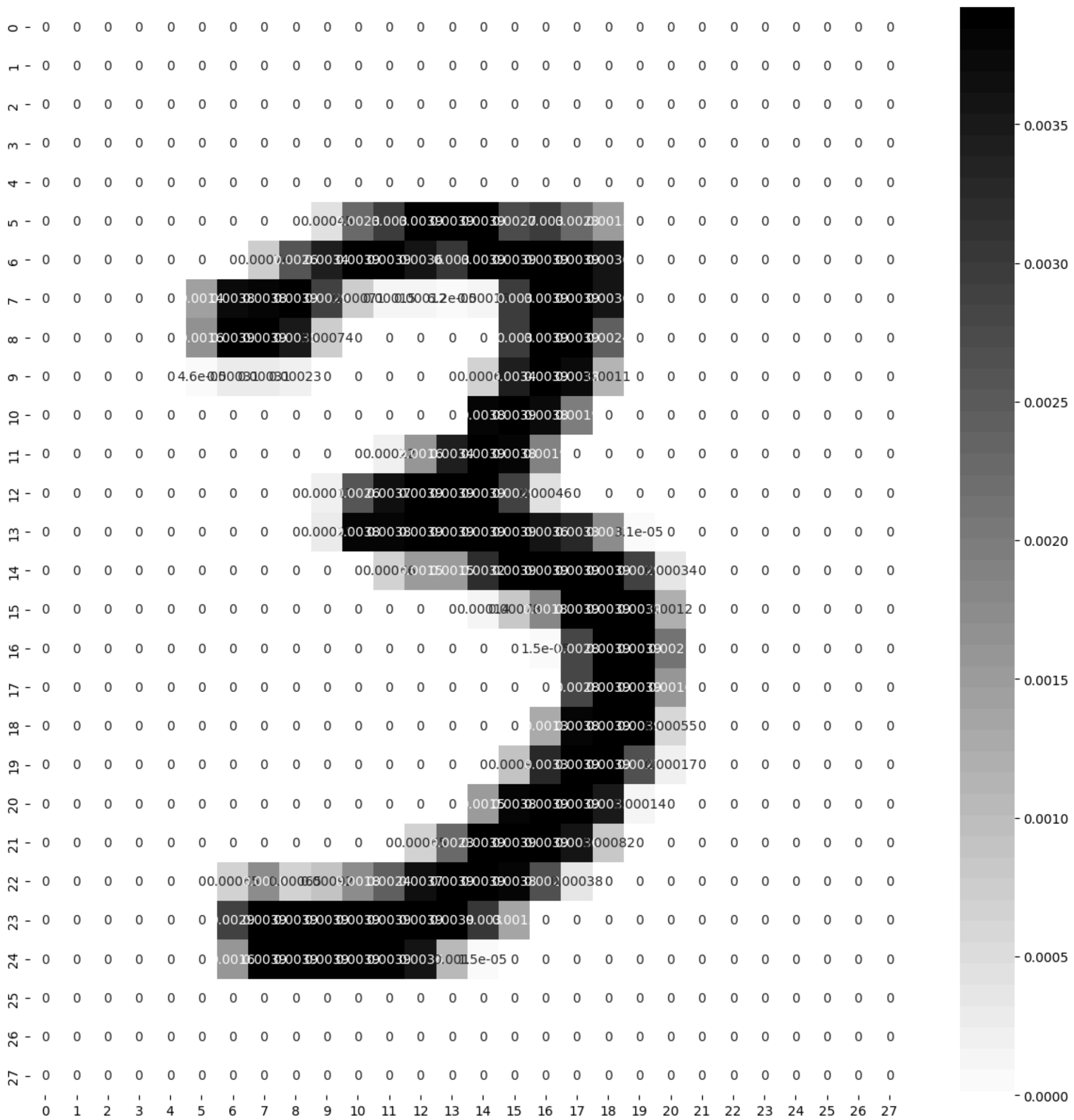
In [12]:
```
x_valid,x_train = x_train[:5000]/255.,x_train[5000:]/255.
y_valid,y_train = y_train[:5000],y_train[5000:]
x_test = x_test/255.
```

In [14]:
```
plt.imshow(x_valid[200],cmap='binary')
plt.show()
```



In [15]:
```
plt.figure(figsize=(15,15))
sns.heatmap(x_train[0],annot=True,cmap='binary')
```

Out[15]:
```
<Axes: >
```



In [18]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Flatten,BatchNormalization,ReLU
from tensorflow.keras.optimizers import Adam
```

In [34]:
```
def create_model_without_bn():
    model = Sequential([
        Flatten(input_shape=[28,28],name = 'inputLayer'),
        Dense(256,activation="relu",name="hiddenLayer1"),
        Dense(128,activation='relu',name="hiddenLayer2"),
        Dense(10,activation='softmax',name="outputLayer")
    ])
    model.compile(optimizer=Adam(learning_rate=0.001),
                loss = 'sparse_categorical_crossentropy',
                metrics = ['accuracy'])
    return model
```

In [35]:
```
model_without_bn = create_model_without_bn()
model_without_bn.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
inputLayer (Flatten)	(None, 784)	0
hiddenLayer1 (Dense)	(None, 256)	200,960
hiddenLayer2 (Dense)	(None, 128)	32,896
outputLayer (Dense)	(None, 10)	1,290

Total params: 235,146 (918.54 KB)

Trainable params: 235,146 (918.54 KB)

Non-trainable params: 0 (0.00 B)

In [36]:
```
layer1 = model_without_bn.layers[1]
layer1.name
```

Out[36]: 'hiddenLayer1'

In [37]:
```
layer1.get_weights()
```

Out[37]:
```
[array([[ 0.00024679,  0.01895618, -0.03377344, ...,  0.0686481 ,
         -0.02457647, -0.06941981],
        [ 0.00959727, -0.04944124,  0.07086723, ...,  0.01559485,
         -0.00149529,  0.06366555],
        [ 0.03715939,  0.06979208, -0.06463815, ..., -0.02731165,
          0.06465426,  0.00370755],
        ...,
        [ 0.05847649,  0.00189476, -0.07038119, ...,  0.07156751,
         -0.00198147, -0.05277468],
        [ 0.00424322,  0.03120036,  0.05713306, ..., -0.0231708 ,
          0.05896318,  0.06744424],
        [-0.07490488,  0.03696477,  0.00724544, ..., -0.03196631,
         -0.0306811 ,  0.02167708]], dtype=float32),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0.], dtype=float32)]
```

In [39]:
```
Epochs = 30
validationset = (x_valid,y_valid)

history  = model_without_bn.fit(x_train,y_train,epochs = Epochs,validation_data
```

```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.6208 - loss: 1.2019 - va
l_accuracy: 0.8916 - val_loss: 0.3939
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8939 - loss: 0.3700 - va
l_accuracy: 0.9058 - val_loss: 0.3397
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9102 - loss: 0.3047 - va
l_accuracy: 0.9152 - val_loss: 0.2928
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9215 - loss: 0.2666 - va
l_accuracy: 0.9248 - val_loss: 0.2602
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9334 - loss: 0.2290 - va
l_accuracy: 0.9314 - val_loss: 0.2345
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9442 - loss: 0.1930 - va
l_accuracy: 0.9416 - val_loss: 0.1967
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9501 - loss: 0.1738 - va
l_accuracy: 0.9434 - val_loss: 0.1834
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9542 - loss: 0.1507 - va
l_accuracy: 0.9506 - val_loss: 0.1647
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9622 - loss: 0.1272 - va
l_accuracy: 0.9528 - val_loss: 0.1497
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9635 - loss: 0.1184 - va
l_accuracy: 0.9562 - val_loss: 0.1453
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9672 - loss: 0.1066 - va
l_accuracy: 0.9586 - val_loss: 0.1336
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9732 - loss: 0.0898 - va
l_accuracy: 0.9626 - val_loss: 0.1172
Epoch 13/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9721 - loss: 0.0905 - va
l_accuracy: 0.9642 - val_loss: 0.1146
Epoch 14/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9759 - loss: 0.0779 - va
l_accuracy: 0.9630 - val_loss: 0.1257
Epoch 15/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9785 - loss: 0.0699 - va
l_accuracy: 0.9652 - val_loss: 0.1088
Epoch 16/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9794 - loss: 0.0646 - va
l_accuracy: 0.9688 - val_loss: 0.1025
Epoch 17/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9813 - loss: 0.0613 - va
l_accuracy: 0.9690 - val_loss: 0.1019
Epoch 18/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9828 - loss: 0.0550 - va
l_accuracy: 0.9694 - val_loss: 0.1105
Epoch 19/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9842 - loss: 0.0519 - va
l_accuracy: 0.9712 - val_loss: 0.0984
Epoch 20/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9854 - loss: 0.0465 - va
l_accuracy: 0.9706 - val_loss: 0.0952
```

```
Epoch 21/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9889 - loss: 0.0379 - va
l_accuracy: 0.9704 - val_loss: 0.1033
Epoch 22/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9877 - loss: 0.0381 - va
l_accuracy: 0.9702 - val_loss: 0.0988
Epoch 23/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9893 - loss: 0.0353 - va
l_accuracy: 0.9746 - val_loss: 0.0900
Epoch 24/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9907 - loss: 0.0311 - va
l_accuracy: 0.9714 - val_loss: 0.0992
Epoch 25/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9912 - loss: 0.0289 - va
l_accuracy: 0.9752 - val_loss: 0.0964
Epoch 26/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9916 - loss: 0.0257 - va
l_accuracy: 0.9720 - val_loss: 0.1028
Epoch 27/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9932 - loss: 0.0243 - va
l_accuracy: 0.9756 - val_loss: 0.0936
Epoch 28/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9939 - loss: 0.0215 - va
l_accuracy: 0.9710 - val_loss: 0.0995
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9950 - loss: 0.0192 - va
l_accuracy: 0.9708 - val_loss: 0.1050
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9942 - loss: 0.0190 - va
l_accuracy: 0.9758 - val_loss: 0.0931
```

In [42]:
```
history.params
```

Out[42]:
```
{'verbose': 'auto', 'epochs': 30, 'steps': 1563}
```

In [43]:
```
pd.DataFrame(history.history)
```

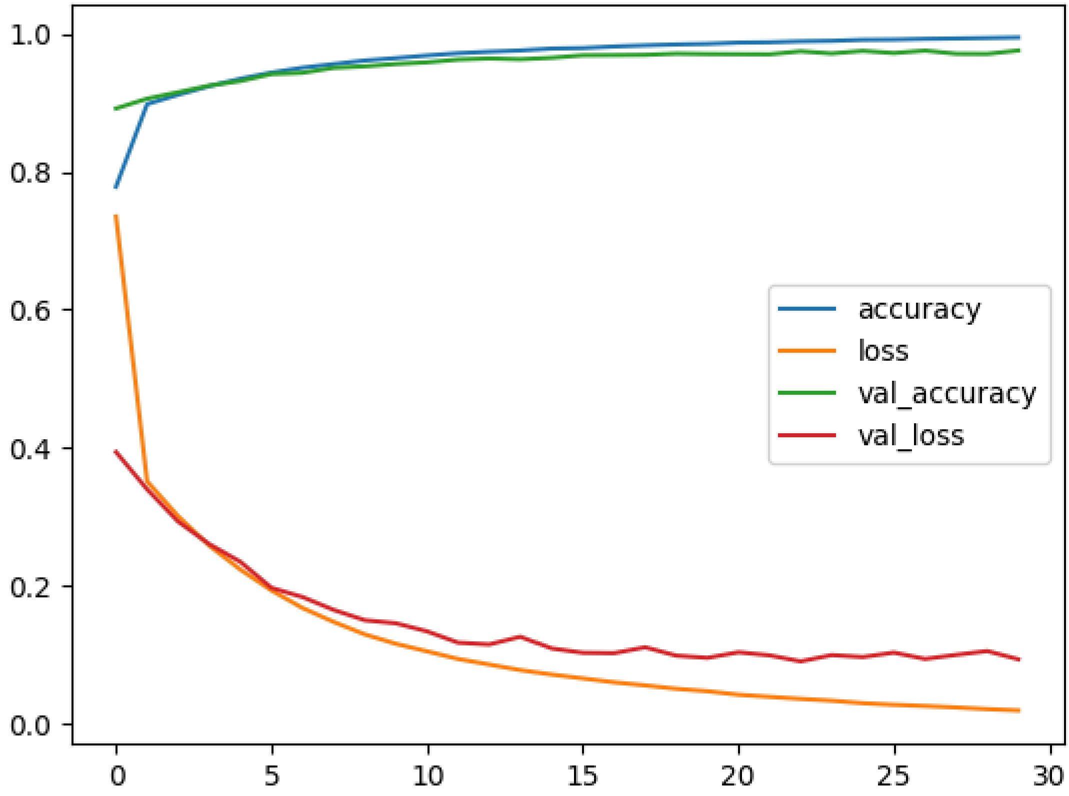
Out[43]:

	accuracy	loss	val_accuracy	val_loss
0	0.77814	0.734837	0.8916	0.393883
1	0.89794	0.351283	0.9058	0.339719
2	0.91156	0.300605	0.9152	0.292824
3	0.92402	0.257924	0.9248	0.260173
4	0.93472	0.223239	0.9314	0.234498
5	0.94370	0.192980	0.9416	0.196736
6	0.95080	0.167460	0.9434	0.183362
7	0.95598	0.147306	0.9506	0.164726
8	0.96124	0.129206	0.9528	0.149742
9	0.96468	0.115553	0.9562	0.145345
10	0.96852	0.104629	0.9586	0.133631
11	0.97176	0.093529	0.9626	0.117221
12	0.97400	0.085484	0.9642	0.114638
13	0.97580	0.077423	0.9630	0.125736
14	0.97858	0.070896	0.9652	0.108784
15	0.97930	0.065332	0.9688	0.102515
16	0.98158	0.059658	0.9690	0.101945
17	0.98320	0.055195	0.9694	0.110516
18	0.98454	0.050270	0.9712	0.098384
19	0.98542	0.046739	0.9706	0.095155
20	0.98722	0.041644	0.9704	0.103281
21	0.98766	0.038588	0.9702	0.098776
22	0.98908	0.035614	0.9746	0.090047
23	0.98974	0.033035	0.9714	0.099173
24	0.99128	0.029360	0.9752	0.096361
25	0.99162	0.027049	0.9720	0.102797
26	0.99264	0.025291	0.9756	0.093577
27	0.99322	0.023180	0.9710	0.099486
28	0.99390	0.020765	0.9708	0.105004
29	0.99466	0.018906	0.9758	0.093074

In [44]:

```
pd.DataFrame(history.history).plot()
```

Out[44]: <Axes: >



In [45]:
```
x_new = x_test[5:12]
```

In [47]:
```
actual = y_test[5:12]
actual
```

Out[47]:
```
array([1, 4, 9, 5, 9, 0, 6], dtype=uint8)
```

In [48]:
```
y_prob = model_without_bn.predict(x_new)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step
```

In [49]:
```
y_prob.round(3)
```

Out[49]:
```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.]], dtype=float32)
```

In [50]:
```
y_pred = np.argmax(y_prob,axis=-1)
y_pred
```

Out[50]:
```
array([1, 4, 3, 5, 9, 0, 6], dtype=int64)
```

In [51]:
```
actual
```

Out[51]:
```
array([1, 4, 9, 5, 9, 0, 6], dtype=uint8)
```

In [58]:
```
# Define the model with batch normalization
def create_model_with_bn():
    model = Sequential([
```

```
        Flatten(input_shape=(28, 28)),
        BatchNormalization(),
        Dense(256,activation="relu"),
        BatchNormalization(),
        Dense(128,activation="relu"),
        BatchNormalization(),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=Adam(learning_rate=0.001),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# Train the model
model_with_bn = create_model_with_bn()
history_with_bn = model_with_bn.fit(x_train, y_train,
                                    epochs=15,
                                    batch_size=64,
                                    validation_data=(x_test, y_test))
```

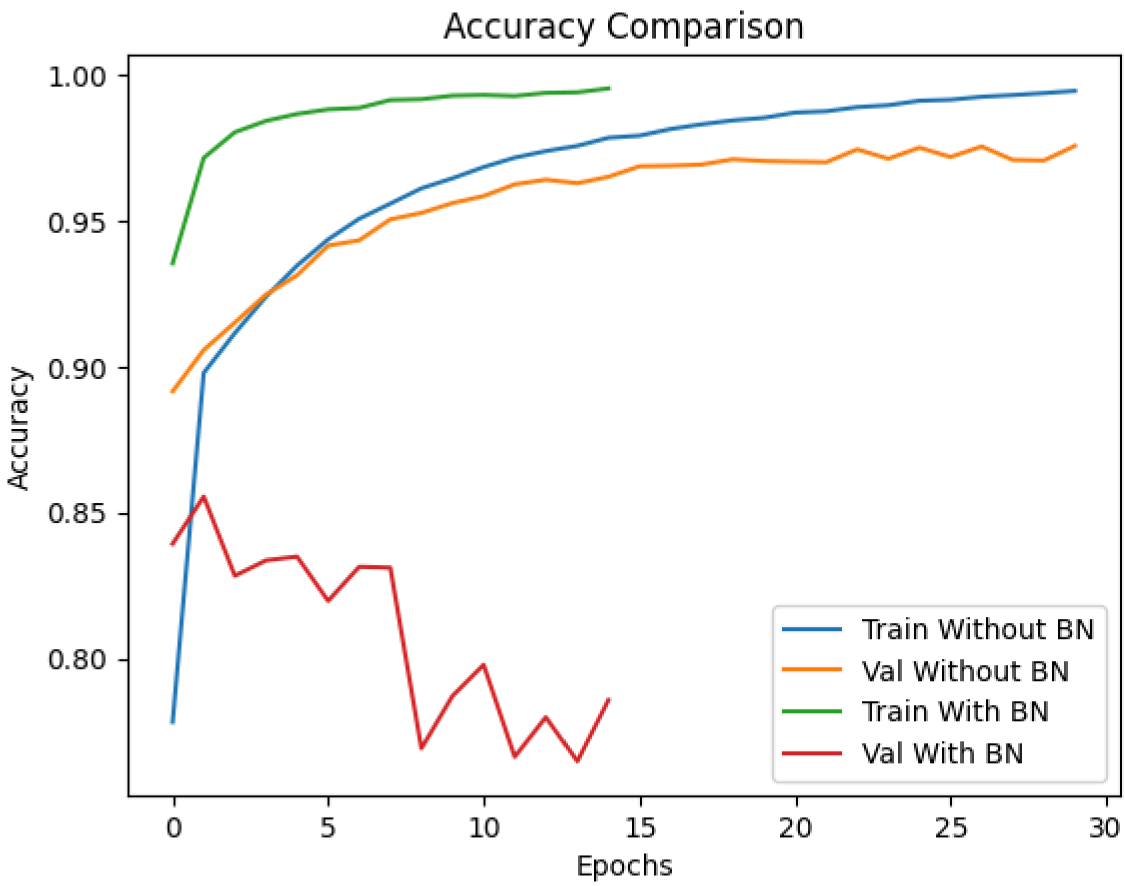
```
Epoch 1/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.8886 - loss: 0.3707 - val_
accuracy: 0.8393 - val_loss: 75.1736
Epoch 2/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9721 - loss: 0.0919 - val_
accuracy: 0.8555 - val_loss: 86.2978
Epoch 3/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9815 - loss: 0.0578 - val_
accuracy: 0.8283 - val_loss: 161.3035
Epoch 4/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9858 - loss: 0.0455 - val_
accuracy: 0.8336 - val_loss: 146.3367
Epoch 5/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9881 - loss: 0.0352 - val_
accuracy: 0.8349 - val_loss: 162.6207
Epoch 6/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9894 - loss: 0.0309 - val_
accuracy: 0.8197 - val_loss: 197.8586
Epoch 7/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9879 - loss: 0.0348 - val_
accuracy: 0.8314 - val_loss: 194.8844
Epoch 8/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9922 - loss: 0.0240 - val_
accuracy: 0.8311 - val_loss: 217.2879
Epoch 9/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9927 - loss: 0.0226 - val_
accuracy: 0.7691 - val_loss: 322.8061
Epoch 10/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9933 - loss: 0.0210 - val_
accuracy: 0.7871 - val_loss: 300.1436
Epoch 11/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.9945 - loss: 0.0162 - val_
accuracy: 0.7978 - val_loss: 303.1212
Epoch 12/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9936 - loss: 0.0191 - val_
accuracy: 0.7662 - val_loss: 279.5966
Epoch 13/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9944 - loss: 0.0160 - val_
accuracy: 0.7798 - val_loss: 295.4492
Epoch 14/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9940 - loss: 0.0182 - val_
accuracy: 0.7647 - val_loss: 301.6129
Epoch 15/15
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9953 - loss: 0.0134 - val_
accuracy: 0.7857 - val_loss: 326.0355
```

In [59]:

```python
import matplotlib.pyplot as plt

# Evaluate models
print("Model Without Batch Normalization:")
model_without_bn.evaluate(x_test, y_test)

print("Model With Batch Normalization:")
model_with_bn.evaluate(x_test, y_test)

# Plot accuracy comparison
plt.plot(history.history['accuracy'], label='Train Without BN')
plt.plot(history.history['val_accuracy'], label='Val Without BN')
plt.plot(history_with_bn.history['accuracy'], label='Train With BN')
plt.plot(history_with_bn.history['val_accuracy'], label='Val With BN')
```

```
plt.title('Accuracy Comparison')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Model Without Batch Normalization:
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8845 - loss: 80.7636
Model With Batch Normalization:
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7681 - loss: 334.2075
```

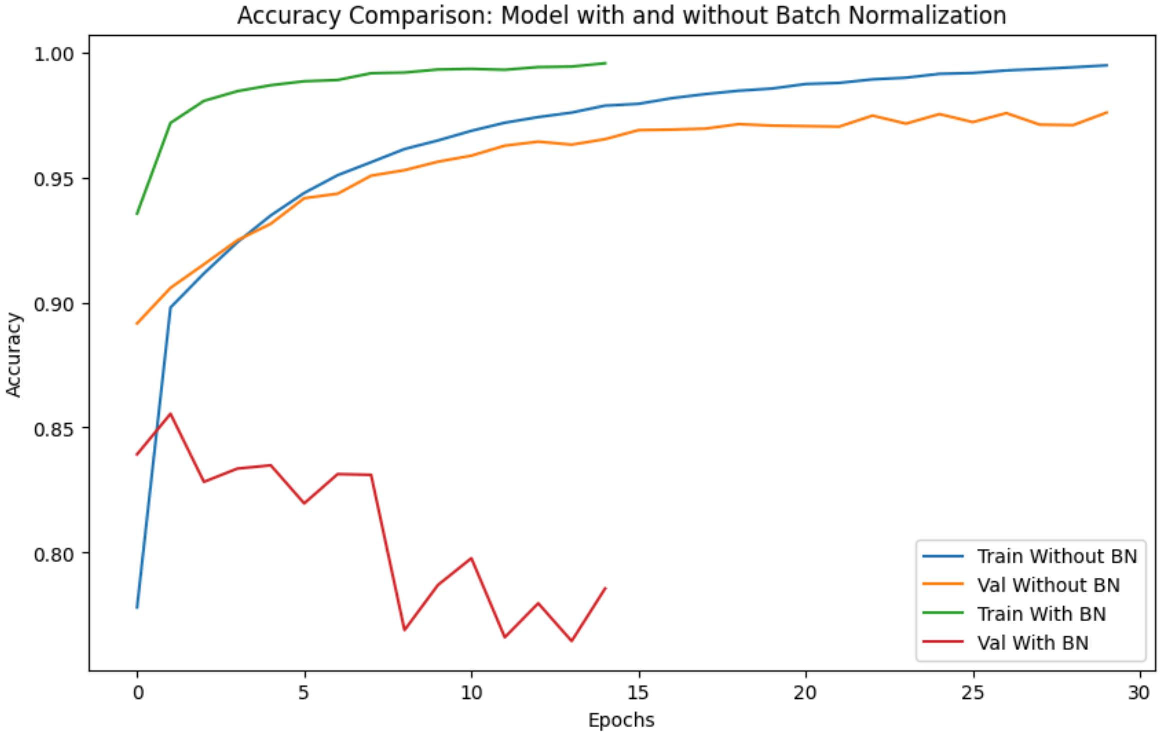


In [60]:

```
import matplotlib.pyplot as plt

# Evaluate models
print("Model Without Batch Normalization:")
loss_without_bn, accuracy_without_bn = model_without_bn.evaluate(x_test, y_test)
print(f"Test Accuracy without BN: {accuracy_without_bn:.4f}")
print(f"Test Loss without BN: {loss_without_bn:.4f}")

print("Model With Batch Normalization:")
loss_with_bn, accuracy_with_bn = model_with_bn.evaluate(x_test, y_test)
print(f"Test Accuracy with BN: {accuracy_with_bn:.4f}")
print(f"Test Loss with BN: {loss_with_bn:.4f}")

# Plot accuracy comparison
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Train Without BN')
plt.plot(history.history['val_accuracy'], label='Val Without BN')
plt.plot(history_with_bn.history['accuracy'], label='Train With BN')
plt.plot(history_with_bn.history['val_accuracy'], label='Val With BN')

# Add title and labels
plt.title('Accuracy Comparison: Model with and without Batch Normalization')
```

```python
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Show the plot
plt.show()
```

```
Model Without Batch Normalization:
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8844 - loss: 80.7867
Test Accuracy without BN: 0.8981
Test Loss without BN: 69.9988
Model With Batch Normalization:
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.7681 - loss: 334.1991
Test Accuracy with BN: 0.7857
Test Loss with BN: 326.0353
```



In [ ]:

Q3. Experimentation and Analysis Experimenting with Batch Sizes Train the model with varying batch sizes (e.g., 16, 64, 256) and observe:

Larger batch sizes may result in smoother gradients and faster convergence. Smaller batch sizes may provide noisier updates but help escape local minima. Advantages of Batch Normalization: Improved Training Stability: Normalized inputs ensure that activations do not diverge. Regularization Effect: Acts as a form of implicit regularization, reducing overfitting. Robust to Initialization: Reduces dependency on careful weight initialization. Limitations: Dependency on Mini-Batches: Batch normalization requires a sufficiently large mini-batch for accurate statistics. Training Overhead: Additional computations for normalization and maintaining moving averages increase training time. Not Always Effective: In certain architectures (e.g., recurrent neural networks), batch normalization may not yield significant improvements, prompting alternatives like Layer Normalization.