Hello. Today, I will be talking about how to develop neural networks for recognizing handwritten digits using the MNIST dataset. My name is Roshni Kasturi, and I'm looking forward to sharing my insights with you.

AI technologies are revolutionizing various industries through object detection, facilitating applications in autonomous vehicles, facial recognition, and security systems.

This presentation aims to detail the development of a neural network model for recognizing handwritten digits, focusing on data preparation, model architecture, training, and evaluation. Today, I will focus on the MNIST dataset, a foundational benchmark for training machine learning models.

The MNIST dataset comprises 70,000 grayscale images of handwritten digits, each measuring 28 by 28 pixels. It includes 60,000 images for training and 10,000 for testing, with each image labeled by its respective digit. This dataset is well-known in the machine learning community as a foundational benchmark for computer vision (Deng, 2012).

To effectively assess our model, it's essential to divide the data into training and validation sets. I chose an 80-20 split for the training data, ensuring a separate validation set to monitor the model's performance. This division is vital for avoiding overfitting and ensuring the model generalizes well to unseen data.

By analyzing the distribution of digits in our training and validation sets, we can assess how balanced the data is. Visualizing a sample from the dataset helps provide context for the information we're working with.

- **Model Evaluation**: The validation set offers an impartial assessment of the model's performance during training. It allows us to track how well the model is learning and aids in fine-tuning hyperparameters.

  Hyperparameter Tuning: The validation set plays a key role in adjusting hyperparameters without affecting the test data, ensuring that the final test set evaluation is accurate and unbiased (Goodfellow et al., 2016).

- **Preventing Overfitting**: Using a separate validation set helps identify overfitting. If the model performs well on training data but poorly on validation data, it indicates that the model is overfitting.

Our neural network architecture consists of an input layer with 784 neurons, each representing a pixel, a hidden layer with 128 neurons using the ReLU activation function, and an output layer with 10 neurons corresponding to the digit classes. This structure strikes a balance between complexity and performance.

The input layer captures pixel data, the hidden layers extract features, and the output layer generates classification probabilities (Russell & Norvig, 2020).

Activation functions are essential for the performance of neural networks. In this model, I employed the ReLU (Rectified Linear Unit) activation function in the hidden layers. ReLU introduces non-linearity, enabling the network to learn complex patterns. For the output layer, I used the Softmax activation function, which is appropriate for multi-class classification tasks, as it transforms the logits into probabilities, helping to identify the most likely class for each input image.

Some benefits of using ReLU are:

- The non-linear nature of ReLU enables the model to learn complex patterns.
- It allows for efficient computation and helps address the vanishing gradient problem (Nair & Hinton, 2010).
- ReLU is simple to implement and proven effective in practice.

This loss function (i.e. Categorical Crossentropy) is utilized for multi-class classification tasks. It quantifies the discrepancy between the true labels and the predicted probabilities. The goal during training is to minimize this loss, which improves the model's performance.

The formula for categorical crossentropy is expressed as shown in the slide.

These parameters were selected through initial experimentation and a review of the literature to optimize the balance between model convergence speed and accuracy.

Choosing 20 Epochs was somewhat arbitrary, but based on some empirical findings, it is appropriate:

- Balanced Training Duration - Training for too few epochs can result in underfitting, where the model fails to adequately learn from the training data. On the other hand, training for too many epochs may lead to overfitting, where the model picks up on noise and specific details of the training set, performing poorly on new data. Twenty epochs often provide a suitable balance for various datasets, including MNIST.

- Empirical Success - In many machine learning tasks, especially with simpler datasets like MNIST, 20 epochs have proven sufficient for model convergence. Research and previous studies involving the MNIST dataset frequently use similar epoch ranges (for example, between 10 and 30 epochs) to achieve good results (Ciresan et al., 2012).
- Validation Monitoring - Utilizing a moderate number of epochs facilitates techniques like early stopping, where training can be stopped if validation performance does not improve after a set number of epochs (Patel et al., 2016). This approach helps prevent overfitting while still providing adequate time for the model to learn.

The batch size was 32. The Learning rate was set to default, a few reasons are discussed herein:

- Proven Effectiveness of Defaults: The Adam optimizer's default learning rate of 0.001 has been validated by extensive empirical research, demonstrating strong performance across various tasks and datasets, including the MNIST dataset, which is frequently used for benchmarking algorithms (Deng, 2012).
- Simplifies Model Development: For those new to machine learning or when rapidly prototyping models, default settings allow practitioners to concentrate on the architecture and overall workflow rather than spending time on hyperparameter tuning. This is especially useful in educational settings or initial experiments (Goodfellow et al., 2016).
- Baseline Performance: Starting with default parameters establishes a baseline performance level. This enables users to make informed decisions about

adjusting the learning rate based on initial outcomes, a common practice in model evaluation (Loshchilov & Hutter, 2016).

- Rapid Prototyping: During the exploratory phase of machine learning projects, utilizing default parameters allows for quicker iterations. This facilitates a swift assessment of the model's capabilities before delving into more detailed hyperparameter tuning (Bergstra & Bengio, 2012).

Layer Combinations for Feature Extraction: The CNN architecture is designed to progressively extract features from input images (Goodfellow et al., 2016).

- **Convolutional Layers (Conv2D):** These layers utilize learnable filters to identify patterns in the input images. By applying multiple filters, the network captures a variety of features, such as edges, corners, and textures (Chollet, 2017).
- **Pooling Layers (MaxPool2D):** Pooling layers reduce the spatial dimensions of feature maps, enhancing the network's robustness to minor variations in input. Max pooling selects the maximum value from a pooling window, retaining the most significant features (Goodfellow et al., 2016).
- **Flattening Layer (Flatten):** This layer transforms the multi-dimensional feature maps into a single vector, preparing the data for fully connected layers (Chollet, 2017).

This combination of convolutional and pooling layers enables the network to learn hierarchical data representations, starting with simple features and advancing to more complex ones (Goodfellow et al., 2016). A filter size of 3x3 was selected as it strikes a good balance between computational efficiency and performance.

Regularization techniques help prevent overfitting, which occurs when a model learns the training data too well, leading to poor performance on unseen data (Goodfellow et al., 2016). Early stopping via the EarlyStopping technique was used herein, and the patience was provided as 2 to achieve this goal. In the 20 epochs that were run, early stopping was not required since there was a progressive validation loss achieved during training.

Optimization algorithms adjust the model's weights during training to minimize the loss function, thereby enhancing prediction accuracy (Goodfellow et al., 2016).

- Loss Function (categorical_crossentropy): measures the difference between predicted probabilities and true labels. The goal is to minimize this loss during training. Categorical cross-entropy is suitable for multi-class classification tasks, such as digit recognition (Chollet, 2017).
- Optimizer (adam): a widely-used optimization algorithm recognized for its efficiency and effectiveness across various deep learning tasks. It adaptively adjusts the learning rate for each parameter, promoting faster convergence (Goodfellow et al., 2016).
- Metrics (accuracy): This metric assesses the model's performance by calculating the percentage of correctly classified images.

The integration of an appropriate loss function, an efficient optimizer, and relevant metrics facilitates effective training and evaluation of the model (Goodfellow et al., 2016).

One of the challenges I faced was with my initial use of sparse categorical_crossentropy, which was not successful due to the one-hot encoding I had done on the variable storing the labels, so I pivoted. In summary, this code employs a well-structured CNN architecture for feature extraction, utilizes regularization to mitigate overfitting, and applies suitable optimization techniques for efficient training.

Let's compare the training vs validation performance. On the left, we have the Training Loss Vs Validation Loss plot. The following observations have been made:

- Both losses are generally decreasing over the epochs (i.e. model is learning and improving its predictions)
- Both lines appear to be converging to a lower value in the later epochs (i.e. model is approaching its optimal performance)
- There exists a small gap between the two losses (i.e. model is generalizing well to unseen/validation data, and is not overfitting to the training data)

On the right, we have the Training Accuracy Vs Validation Accuracy plot. It can be observed that:

- Both accuracies are generally increasing and converging over the epochs with an upward trend (i.e. model is learning patterns in data effectively)
- Both lines are reaching high accuracy values (i.e. model is performing well on seen and on unseen data)
- Both lines are in alignment - validation accuracy is following training accuracy (i.e. no signs of over or underfitting, and no plateau in validation accuracy - or an inverse relationship to training accuracy)

Let’s now examine the generated classification report. We can see a 97% accuracy, while the macro and weighted averages for precision, recall, and F1-score are also 97%. The consistency across these averages suggests that the model is performing well across all classes (i.e. there is no significant bias towards any specific digit).

Deep diving into per-class performance:

- Precision is between 95-97% for most classes, with digit 2 having a slightly lower precision at 95%
- Recall is in the high 90s for most classes, with digit 7 slightly suffering at 93%
- F1-scoreis also in the high 90s for most classes, with digits 2, 7, and 9 having lower scores
- Finally, support is relatively balanced

To recap, the confusion matrix basics are that the rows represent true labels, the columns represent predicted labels, the diagonal elements of the matrix represent correctly classified instances for each digit, and the off-diagonal elements represent misclassifications. Now let’s examine this confusion matrix:

- There are large numbers in the diagonal, which means model is achieving high accuracy in classifying most digits
- There are off diagonal numbers, albeit very small, which means low rate of misclassifications
- As we speak about misclassifications, let’s look at a few that the model made:

1. Digit 4 was misclassified as 9, 21 times. One may think - could this be due to the way digit 4 is handwritten differently by different people?

2. Digit 7 was misclassified as 9, 26 times
3. Digit 3 was misclassified as 5, 12 times
4. Digit 9 was misclassified as 4, 10 times (which connects back to 1st misclassification that we discussed).

In sum total, the results demonstrate a strong balance between training and validation performance. Additionally, tools at our disposal, such as the confusion matrix, offered insights into the model's performance across various categories, indicating areas where improvements could be made.

In this project, a neural network model was created that can recognize handwritten numerical digits from the MNIST dataset, which was an excellent benchmark for the model testing. Key emphasis was placed on its architecture, training process, and evaluation methods. It was also emphasized that having and using a separate validation set is vital for assessing model performance in parallel to the training set. Producing an artifact as such results in reinforcement of neural networks concepts and their components, via practical implementation using Keras and TensorFlow.
Let’s discuss some potential improvements that could result in further optimization of the model performance. The first being experimentation with different CNN architectures / variations, and the second being experimentation with hyperparameter tuning (such as with no. of filters / kernel size / no. of neurons in dense layers).

This activity taught me the significance of data preparation, the effects of various neural network architectures, and the importance of thorough model evaluation. I recall getting

a very high loss value when I finally ran the model on the test data, and I was surprised to see that because the model performed very well on training and validation data. I soon realized that I was referencing an old variable (x_test), vs the new one (x_test_enhanced) that had an added channel dimension. This loss feedback helped me rectify the error.

Thank you for your time!

Finally, I have provided references for the resources and literature that informed my research, approach, and implementation of the neural network model.