Building a CNN for MNIST Handwritten Digit Classification

Introduction

Welcome! In this assignment, you will build a Convolutional Neural Network (CNN) to classify handwritten digits from the famous MNIST dataset. This dataset is a classic in the field of computer vision and provides a great starting point for understanding image classification with deep learning.

This notebook is structured to guide you step-by-step through the process. You will load the data, preprocess it, define a CNN model, train it, and evaluate its performance. Throughout the assignment, you will have opportunities to experiment and deepen your understanding of the concepts.

Remember to:

- **Read all instructions carefully.**
- **Execute the code cells in order.**
- **Fill in the missing code sections marked as "Students: Fill in the blanks".**
- **Answer the reflection questions in the designated Markdown cell.**
- **Experiment and explore!** Change parameters, layers, and observe the effects.

Let's get started and build our MNIST digit classifier!

Section 1: Setting Up - Imports

Before we dive into building our CNN, we need to import the necessary libraries. These libraries provide pre-built tools and functions that will make our work much easier.

Instructions:

1. **Carefully review the code cell below.** It imports libraries from TensorFlow and Keras, which are powerful frameworks for building and training neural networks.
2. **Execute the code cell by selecting it and pressing [Shift + Enter] (or the "Run" button).**
3. **Ensure there are no error messages after running the cell.** If you encounter errors, double-check that you have TensorFlow and Keras installed in your environment.

Explanation of Imports:

- **`tensorflow as tf and keras:`** TensorFlow is the main deep learning framework, and Keras is its high-level API that simplifies building and training models. We import TensorFlow as `tf` and Keras directly for easy access to their functionalities.

- **`from tensorflow.keras import layers`:** This imports the `layers` module from Keras, which provides various layers for building neural networks (like convolutional layers, dense layers, etc.).
- **`from tensorflow.keras.datasets import mnist`:** This imports the MNIST dataset directly from Keras datasets. This is very convenient for loading and using the MNIST data.
- **`from tensorflow.keras.utils import to_categorical`:** This imports the `to_categorical` function, which we will use to perform one-hot encoding of our labels.

Section 2: Data Loading and Preprocessing

In this section, we will load the MNIST dataset and prepare it for training our CNN model. Preprocessing steps are crucial to ensure our data is in the right format for the model to learn effectively.

Instructions:

1. **Read through the code in the cell below.** Understand how it loads the MNIST dataset and what preprocessing steps are applied.
2. **Execute the code cell.**
3. **Examine the comments in the code** to understand each preprocessing step in detail.

```
# Cell 1: Imports
# Importing necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import numpy as np
```

```
pip install tensorflow
```

```
Requirement already satisfied: tensorflow in /opt/anaconda3/lib/python3.12/site-
Requirement already satisfied: absl-py>=1.0.0 in /opt/anaconda3/lib/python3.12/s
Requirement already satisfied: astunparse>=1.6.0 in /opt/anaconda3/lib/python3.1
Requirement already satisfied: flatbuffers>=24.3.25 in /opt/anaconda3/lib/python
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /opt/anaco
Requirement already satisfied: google-pasta>=0.1.1 in /opt/anaconda3/lib/python3
Requirement already satisfied: libclang>=13.0.0 in /opt/anaconda3/lib/python3.12
Requirement already satisfied: opt-einsum>=2.3.2 in /opt/anaconda3/lib/python3.1
Requirement already satisfied: packaging in /opt/anaconda3/lib/python3.12/site-p
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.2
Requirement already satisfied: requests<3,>=2.21.0 in /opt/anaconda3/lib/python3
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.12/site-
Requirement already satisfied: six>=1.12.0 in /opt/anaconda3/lib/python3.12/site
Requirement already satisfied: termcolor>=1.1.0 in /opt/anaconda3/lib/python3.12
```

```
Requirement already satisfied: typing-extensions>=3.6.6 in /opt/anaconda3/lib/py
Requirement already satisfied: wrapt>=1.11.0 in /opt/anaconda3/lib/python3.12/si
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /opt/anaconda3/lib/python3
Requirement already satisfied: tensorboard<2.19,>=2.18 in /opt/anaconda3/lib/pyt
Requirement already satisfied: keras>=3.5.0 in /opt/anaconda3/lib/python3.12/sit
Requirement already satisfied: numpy<2.1.0,>=1.26.0 in /opt/anaconda3/lib/python
Requirement already satisfied: h5py>=3.11.0 in /opt/anaconda3/lib/python3.12/sit
Requirement already satisfied: ml-dtypes<0.5.0,>=0.4.0 in /opt/anaconda3/lib/pyt
Requirement already satisfied: wheel<1.0,>=0.23.0 in /opt/anaconda3/lib/python3.
Requirement already satisfied: rich in /opt/anaconda3/lib/python3.12/site-packag
Requirement already satisfied: namex in /opt/anaconda3/lib/python3.12/site-packa
Requirement already satisfied: optree in /opt/anaconda3/lib/python3.12/site-pack
Requirement already satisfied: charset-normalizer<4,>=2 in /opt/anaconda3/lib/py
Requirement already satisfied: idna<4,>=2.5 in /opt/anaconda3/lib/python3.12/sit
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/anaconda3/lib/python3.
Requirement already satisfied: certifi>=2017.4.17 in /opt/anaconda3/lib/python3.
Requirement already satisfied: markdown>=2.6.8 in /opt/anaconda3/lib/python3.12/
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /opt/ana
Requirement already satisfied: werkzeug>=1.0.1 in /opt/anaconda3/lib/python3.12/
Requirement already satisfied: MarkupSafe>=2.1.1 in /opt/anaconda3/lib/python3.1
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in /opt/anaconda3/li
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /opt/anaconda3/lib/pyt
Requirement already satisfied: mdurl~=0.1 in /opt/anaconda3/lib/python3.12/site-
Note: you may need to restart the kernel to use updated packages.
```

```
# Cell 2: Data Loading and Preprocessing
# Load the MNIST dataset
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Add a channel dimension (for grayscale images, it's 1)
x_train = x_train.reshape(-1, 28, 28, 1)
x_test = x_test.reshape(-1, 28, 28, 1)

# One-hot encode the labels
from tensorflow.keras.utils import to_categorical
num_classes = 10
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

Explanation of Data Preprocessing:

- **Loading the MNIST dataset:** `mnist.load_data()` loads the MNIST dataset, which is already split into training and testing sets (`(x_train, y_train), (x_test, y_test)`). `x_train` and `x_test` contain the images (pixel data), and `y_train` and `y_test` contain the corresponding labels (digits 0-9).

- **Normalization:** `x_train = x_train.astype("float32") / 255.0` and `x_test = x_test.astype("float32") / 255.0` normalize the pixel values. Pixel values in images are typically in the range 0-255. Dividing by 255 scales them to the range 0-1. This normalization helps the neural network train faster and more effectively.
- **Adding Channel Dimension:** `x_train = x_train.reshape(-1, 28, 28, 1)` and `x_test = x_test.reshape(-1, 28, 28, 1)` reshape the data to add a channel dimension. Even though MNIST images are grayscale (single channel), CNNs in Keras expect input data to have a channel dimension. We reshape from `(number_of_images, 28, 28)` to `(number_of_images, 28, 28, 1)`. The `-1` in `reshape` means "infer the dimension based on the size of the array."
- **One-Hot Encoding:** `y_train = to_categorical(y_train, num_classes)` and `y_test = to_categorical(y_test, num_classes)` perform one-hot encoding on the labels. Instead of representing the digit '3' as a single number, one-hot encoding converts it into a vector `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`, where the 4th position (index 3) is 'hot' (value 1), and all other positions are 'cold' (value 0). This is a standard way to represent categorical labels for neural networks in multi-class classification problems. `num_classes = 10` specifies that we have 10 classes (digits 0-9).

Section 3: Model Definition - Building the CNN

Now we will define the architecture of our Convolutional Neural Network (CNN). You will be building a sequential model using Keras layers.

Instructions:

1. **Carefully examine the code in the cell below.** Notice the structure of the `keras.Sequential` model.
2. **Fill in the missing parts** marked with `# Students: Fill in the blanks` to complete the model definition.
3. **Experiment!** You are encouraged to try different configurations for the layers, such as changing the number of filters in the convolutional layers, or adding more layers.

```
# Cell 3: Model Definition
# Build the CNN model.  Students: Fill in the missing parts!
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),  # Input layer
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),  # Convolutional l
        layers.MaxPooling2D(pool_size=(2, 2)),  # Max pooling layer 1
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),  # Convolutional l
```

```
        layers.MaxPooling2D(pool_size=(2, 2)),  # Max pooling layer 2
        layers.Conv2D(128, kernel_size=(3, 3), activation="relu"),  # Convolutional
        layers.MaxPooling2D(pool_size=(2, 2)),  # Max pooling layer 3 (optional, but
        layers.Flatten(),  # Flatten layer
        layers.Dropout(0.5),  # Dropout layer
        layers.Dense(num_classes, activation="softmax"),  # Output layer
    ]
)
```

Explanation of Layers:

- **`keras.Input(shape=(28, 28, 1))`:** This is the input layer of our model. It specifies the shape of the input images, which are 28x28 pixels with 1 channel (grayscale).
- **`layers.Conv2D(32, kernel_size=(3, 3), activation="relu")`:** This is a 2D Convolutional layer.
  - `32` : This is the number of filters (also called kernels). Each filter learns to detect specific features in the input image.
  - `kernel_size=(3, 3)` : This defines the size of the convolutional filter as 3x3 pixels.
  - `activation="relu"` : ReLU (Rectified Linear Unit) is the activation function. It introduces non-linearity into the model, allowing it to learn complex patterns.
- **`layers.MaxPooling2D(pool_size=(2, 2))`:** This is a Max Pooling layer.
  - `pool_size=(2, 2)` : It reduces the spatial dimensions of the feature maps by taking the maximum value within each 2x2 window. This helps to reduce the number of parameters, control overfitting, and make the model more robust to small shifts and distortions in the input.
- **`layers.Flatten()`:** This layer flattens the 2D feature maps from the convolutional and pooling layers into a 1D vector. This is necessary to connect the convolutional part of the network to the fully connected (Dense) layers.
- **`layers.Dropout(0.5)`:** This is a Dropout layer.
  - `0.5` : This sets the dropout rate to 50%. During training, this layer randomly sets 50% of the input units to 0 at each update. This is a regularization technique that helps to prevent overfitting.
- **`layers.Dense(num_classes, activation="softmax")`:** This is the output Dense (fully connected) layer.
  - `num_classes` : This is set to 10 because we have 10 classes (digits 0-9).
  - `activation="softmax"` : Softmax activation ensures that the output values are probabilities, and they sum up to 1 across all classes. The output will be a vector of 10

probabilities, where each probability represents the model's confidence that the input image belongs to that specific digit class.

Section 4: Model Compilation - Choosing Loss and Optimizer

Before we can train our model, we need to compile it. Compilation involves choosing an optimizer, a loss function, and metrics to evaluate the model's performance.

Instructions:

1. **Examine the code cell below.** You need to fill in the blanks for the `loss` and `optimizer` parameters in `model.compile()`.
2. **Choose an appropriate loss function and optimizer** for this multi-class classification problem.
3. **In the Markdown cell after the code, explain your choices.** Why are these choices suitable for this task?

```
# Cell 4: Model Compilation
# Students: Choose an appropriate loss fun
ction and optimizer.  Why did you choose these?
# Cell 4: Model Compilation
model.compile(loss="categorical_crossentropy",
              optimizer="adam",
              metrics=["accuracy"])
```

```
Object `these` not found.
```

Explanation of Choices (To be filled by students in the reflection section):

- **Loss Function:** You need to choose a loss function that is appropriate for multi-class classification. Think about what kind of error we are trying to minimize when classifying digits into 10 categories.
- **Optimizer:** You need to choose an optimizer that will efficiently update the model's weights to minimize the loss function. Consider common optimizers used in deep learning.
- **Metrics:** We are using "accuracy" as a metric to evaluate the model's performance. Accuracy is a common metric for classification tasks, representing the percentage of correctly classified images.

Section 5: Model Training - Fitting the Model to the Data

Now it's time to train our CNN model using the training data. Training involves feeding the training data to the model and adjusting its weights to minimize the loss function.

Instructions:

1. **Examine the code cell below.** You need to fill in the blanks for `batch_size` and `epochs` in `model.fit()`.
2. **Choose appropriate values for `batch_size` and `epochs`.**
3. **Run the code cell to start training.** Observe the training progress, especially the loss and accuracy on both the training and validation sets.
4. **Experiment!** Change the `batch_size` and `epochs` and see how it affects the training process and the final performance.

```
# Cell 5: Model Training
# Students: Adjust the batch size and number of epochs.  What happens if you change
model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1) #Studen
# Train the model with adjusted batch size and epochs
model.fit(x_train, y_train, batch_size=128, epochs=15, validation_split=0.1)
```

```
Epoch 1/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.7273 - loss: 0.8641 - va
Epoch 2/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - accuracy: 0.9485 - loss: 0.1770 - va
Epoch 3/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.9616 - loss: 0.1243 - va
Epoch 4/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9698 - loss: 0.1047 - va
Epoch 5/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9716 - loss: 0.0935 - va
Epoch 6/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9759 - loss: 0.0790 - va
Epoch 7/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9793 - loss: 0.0693 - va
Epoch 8/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9815 - loss: 0.0611 - va
Epoch 9/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9808 - loss: 0.0611 - va
Epoch 10/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9835 - loss: 0.0554 - va
Epoch 11/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9844 - loss: 0.0496 - va
Epoch 12/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9850 - loss: 0.0499 - va
Epoch 13/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9867 - loss: 0.0433 - va
Epoch 14/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 15ms/step - accuracy: 0.9868 - loss: 0.0420 - va
Epoch 15/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9876 - loss: 0.0387 - va
Epoch 1/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9888 - loss: 0.0356 - va
Epoch 2/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9890 - loss: 0.0336 - va
Epoch 3/15
```

```
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9905 - loss: 0.0315 - va
Epoch 4/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.9912 - loss: 0.0273 - va
Epoch 5/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9901 - loss: 0.0309 - va
Epoch 6/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9914 - loss: 0.0270 - va
Epoch 7/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9915 - loss: 0.0264 - va
Epoch 8/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9918 - loss: 0.0246 - va
Epoch 9/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9926 - loss: 0.0222 - va
Epoch 10/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9923 - loss: 0.0228 - va
Epoch 11/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9931 - loss: 0.0188 - va
Epoch 12/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 14ms/step - accuracy: 0.9941 - loss: 0.0191 - va
Epoch 13/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9929 - loss: 0.0220 - va
Epoch 14/15
422/422 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9935 - loss: 0.0193 - va
```

Explanation of Training Parameters:

- **`batch_size`:** This determines the number of training samples processed in each mini-batch during training. A larger batch size can speed up training but might require more memory. A smaller batch size can lead to more noisy updates but might generalize better.
- **`epochs`:** One epoch represents one complete pass through the entire training dataset. More epochs can potentially lead to better training but also increase the risk of overfitting, where the model learns the training data too well and performs poorly on unseen data.
- **`validation_split=0.1`:** This reserves 10% of the training data as a validation set. During training, the model's performance is evaluated on this validation set after each epoch. This helps to monitor for overfitting and tune hyperparameters.

Section 6: Model Evaluation - Assessing Performance on Test Data

After training, we need to evaluate our model's performance on the test dataset. This gives us an estimate of how well the model generalizes to unseen data.

Instructions:

1. **Run the code cell below.**
2. **Observe the output.** It will print the test loss and test accuracy.

3. **Think about the results.** Is the test accuracy satisfactory? How does it compare to the training and validation accuracy you observed during training?

```
# Cell 6: Model Evaluation
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test loss: {loss:.4f}")
print(f"Test accuracy: {accuracy:.4f}")
```

Section 7: Reflection and Answers to Questions

This is an important section! Take some time to reflect on what you have learned and answer the following questions in detail. Your thoughtful answers will demonstrate your understanding of the concepts covered in this assignment.

Reflection Questions:

1. **Conv2D Layer:** What is the role of the Conv2D layer? How do the `kernel_size` and the number of filters affect the learning process?
    - Role: Extracts features from images by applying convolutional filters. Kernel size & filters: Smaller kernels (e.g., 3x3) capture detailed features, and more filters detect a variety of features, improving model performance but requiring more computation.
2. **MaxPooling2D Layer:** What is the purpose of the MaxPooling2D layer? How does it contribute to the model's performance?
    - Purpose: Reduces spatial dimensions, decreasing computational cost and overfitting risk. Effect: Removing it may result in overfitting or slower training. Adding more may help reduce dimensions more aggressively.
3. **One-Hot Encoding:** Why do we use one-hot encoding for the labels?
    - Converts class labels into binary vectors for multi-class classification, allowing the model to differentiate classes.
4. **Flatten Layer:** Why do we need the Flatten layer before the Dense layer?
    - Converts 2D feature maps into 1D vectors for the Dense layer.
5. **Optimizer and Loss Function:** What optimizer and loss function did you choose in Cell 4? Explain your choices. Why is categorical cross-entropy a suitable loss function for this task? Why is Adam a good choice of optimiser?
    - Adam optimizer: Adjusts learning rate dynamically for better convergence. Categorical Cross-Entropy: Measures error for multi-class classification tasks.

6. **Batch Size and Epochs:** How did you choose the batch size and number of epochs in Cell 5? What are the effects of changing these parameters? *Hint: Experiment!*
    - Batch Size (128): Larger sizes speed up training but may cause overfitting. Epochs (15): More epochs may improve accuracy but risk overfitting; experimentation is key.
7. **Dropout:** Why is the Dropout layer included in the model?
    - Helps prevent overfitting by randomly deactivating a fraction of neurons during training.
8. **Model Architecture:** Describe the overall architecture of your CNN. How many convolutional layers did you use? How many max pooling layers? What is the final dense layer doing?
    - 3 Conv2D layers, 3 MaxPooling2D layers, 1 Flatten layer, 1 Dropout layer, and 1 Dense output layer with 10 units for classification.
9. **Performance:** What accuracy did you achieve on the test set? Are you happy with the result? Why or why not? If you're not happy, what could you try to improve the performance?
    - A good test accuracy (95%+) is desirable. If lower, experiment with more layers, dropout, or optimizers.

Tips and Explanations:

- **Normalization:** Dividing the pixel values by 255 normalizes them to the range [0, 1]. This is important for training neural networks.
- **Reshaping:** The `reshape` operation adds a channel dimension to the images. For grayscale images, the channel dimension is 1.
- **One-Hot Encoding:** `to_categorical` converts the class labels (0-9) into one-hot encoded vectors.
- **Conv2D Parameters:** The `kernel_size` determines the size of the convolutional filter (e.g., 3x3). The number of filters determines how many different features are learned.
- **MaxPooling2D Parameters:** The `pool_size` determines the size of the pooling window (e.g., 2x2).
- **Optimizer:** The optimizer is the algorithm used to update the model's weights during training.
- **Loss Function:** The loss function measures the error between the model's predictions and the true labels.
- **Batch Size:** The batch size is the number of samples processed in each training iteration.
- **Epochs:** An epoch is one complete pass through the entire training dataset.
- **Dropout:** Dropout is a regularization technique that helps prevent overfitting.

Remember to run each cell to see its output. Experiment with the code and try to understand how

Conclusion and Submission

Congratulations on completing this notebook assignment! You have successfully built and trained a Convolutional Neural Network to classify handwritten digits from the MNIST dataset. You've explored key concepts like convolutional layers, pooling layers, activation functions, optimizers, loss functions, and training procedures. To further solidify your understanding, consider the following:

- **Review your notebook:** Go back through each section, reread the explanations, and make sure you understand the code and the concepts.
- **Experiment further:** Try different CNN architectures, add more layers, change hyperparameters, and see how it affects the performance. Explore other optimizers or loss functions.
- **Reflect on your learning:** Think about the challenges you faced and how you overcame them. What were the most important takeaways for you from this assignment?

Submission Instructions

To submit your assignment:

1. **Save your notebook:** Ensure all your work, including code cells, outputs, and answers to reflection questions, is saved in the notebook.
2. **Print the notebook as a `.pdf` file** and submit it to Canvas.

Deadline: February, 12th