Neural Age Estimation: MRI-Based Brain Age Prediction using Deep Learning

Self Learning Tutorial

Overview and Setup

Github

Here is a link to the project!

Objective

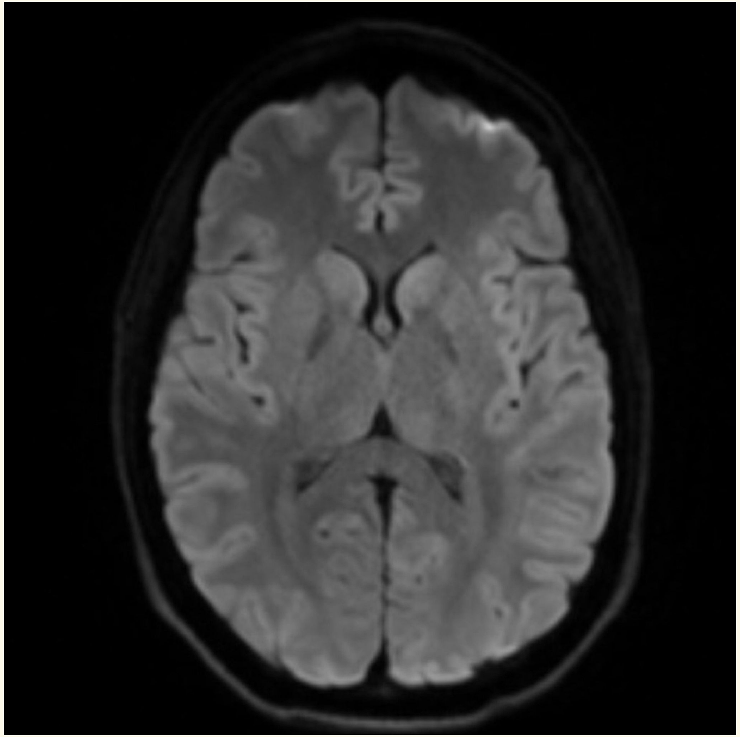
This tutorial aims to predict a person's age by looking at their brain MRI scan.

Data Handling
We will load in and process MRI scans from the OASIS dataset. During this process we will normalize our data and keep track of patient information.

AI Model Structure
We will use a CNN (Convolutional Neural Network) with residual blocks to train our model.

Training Process
We will split our data into training/ test sets. We will apply slight augmentations to our images to improve learning. We will measure our models success and adjust as epochs progress.

Safety Features
We will aim to prevent overfitting, save the best version of the model, and use early stopping once improvements stop.



Open Access Series of Imaging Studies (OASIS)

OASIS-2

Provided by WashU Medicine, this dataset consists of a longitudinal collection of 150 subjects aged 60 to 96 over 373 imaging sessions. The dataset included provides important information and features including patient age.

Usage

We will use this dataset to generate a deep learning model that aims to predict a patient age. We will read in over 1300 3D brain scan images, train a CNN (Convolution Neural Network), and aim to predict the patients age.

Requesting Access

Link to Access

1. Use Agreement

Please read and understand the usage agreement involved with the dataset

2. Fill out Form

Use your UT email and select *OASIS-2: Longitudinal Dementia*

3. Look for Email

A link to the dataset and download will be sent to your email address

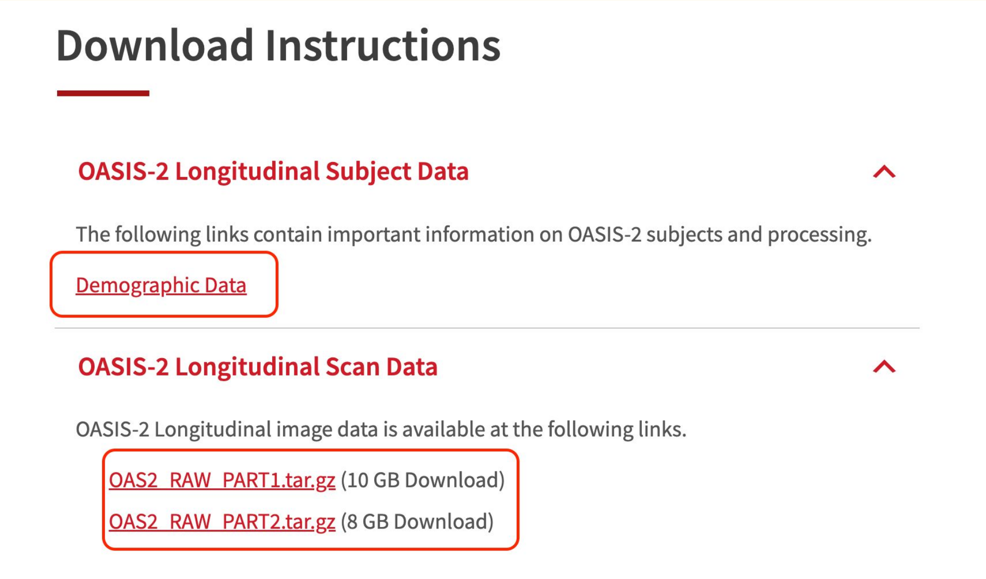
4. Download Data

Download the two parts of the data set as well as the XLSX for patient information

Download the Dataset

1. Subject Data using in conjunction with scan data to predict patient age

2. Multipart download of the image data used to train the Neural Network (~18GB)
3. Link to OASIS-2



Tutorial Outline

Data Loading and Preprocessing

- Read in CSV file
- Load MRI images
- Resample to (64x64x64)
- Normalize image data
- Track patient ID for data splitting

Data Splitting

- Ensure all scans from same patient stay in the same set
- 80/20 training split

Model Architecture

- Convolutional Layer with dropout
- 3 residual blocks with increasing channels
- Fully connected layers
- Global Average Pooling

Data Augmentation

- Random Noise
- Random intensity scaling
- Random horizontal splits
- **Only applied during training

Tutorial Outline (Continued)

Training Process

- Learning rate warm up (3-epochs)
- AdamW optimizer
- MSE loss function
- Gradient Clipping

Training Monitoring

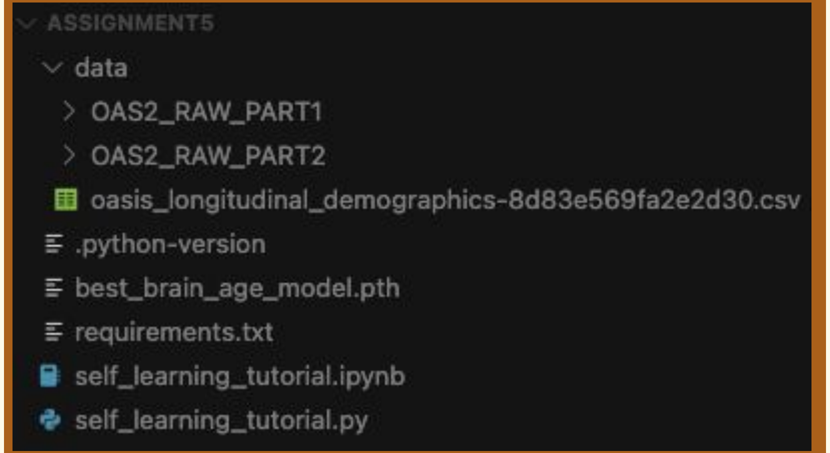
- Track training progress
- Performance metrics output after each epoch

Model Optimization

- Learning rate scheduling
- Early stopping
- Regularization using Dropout, Weight decay, Data augmentation
- Save best Model

Performance Metrics

- Mean Absolute Error in years (MAE)
- Root Mean Square Error (RMSE)
- Training vs. Validation loss
- Age Predictions are denormalized for readability



File Structure and Setup

1. File Setup

The expected file structure is shown to the left. Please use a tool to convert the XLSX file to a CSV to make it easier to work with. You can upload into google sheets and export as CSV. Any method for converting should work here. I have included both a .py and .ipnyb that contain the same functionality. Your end result will save a new .pth file. I have included mine incase you are unable to run the training.

2. System Independent

I have created and ran this project using an M series macbook. I was unable to use Colab due to file size restrictions and timeouts. In order to create support across multiple machines this program will attempt to use CUDA for NVIDIA GPUs, MPS for Apple Silicon, and CPU for all others.

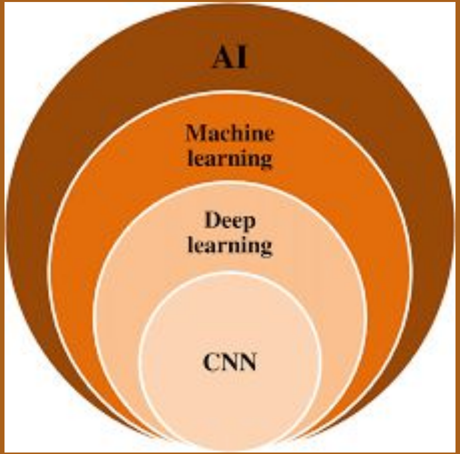
Control Flow Explained

1. Load in our data (MRIs and CSV)
2. Split our data in training and validation set ensuring that we do not have a patient spanning both categories
3. Train our model
4. Training runs through up to 50 epochs
    a. Warm up 3 epochs
    b. Forward pass, calculate loss, backward pass, gradient clipping, optimizer step
    c. Validation after each epoch
    d. Learning rate scheduling
    e. Early stopping check
    f. Save best model
5. Supporting Classes (ResBlock, BrainAgeCNN, BrainAgeDataset)

```
Program Start
│
├─▶ load_data()
│  └─▶ Process images / csv data
│
├─▶ Split data into train/test sets
│
└─▶ train_model()
   ├─▶ Setup model and training
components
   │
   └─▶ Training Loop
      ├─▶ Training phase
      │  └─▶ Process batches
      │
      ├─▶ Validation phase
      │
      └─▶ Model checkpointing
```

Disclaimer

We are dealing with a small dataset relative to the task we are performing. Due to this, it was *extremely* difficult to fine tune hyperparameters without overfitting. Unfortunately system limitations prevented me from training with a larger dataset. If possible, I would encourage you to use a larger OASIS dataset or find another compatible brain scan set.



Code and Explanations

Not all of the code for this project can be found in these slides. However, the code that is paired with this project is well documented. I suggest that you follow along with the code and use this document for supplementary explanations.

load_data()

Packages

- *os*: Path handling and file operations
- *glob*: File pattern matching
- *niable*: Loading and handling medical image files
- *nilearn*: Image resampling and processing
- *pandas*: Reading CSV and data handling
- *numpy*: Numerical operations and array handling
- *tqdm*: Progress bar for data loading

Loading Data

The program looks inside *OAS2_RAW_PART1* and *OAS2_RAW_PART2* using *os* and *glob* to retrieve MRI images. We load in the MRI files using *niable.* We read in the patient data using *pandas.* We count the number of scans and ensure we pull all the scans for each patient as there are a variable amount (2-4) per scan. We create our progress bar using *tqdm* which we will update throughout the loading process. Can take some time to load all of the images ~8-10 minutes.

Image Processing

We use *nilearn* to resample all of our images and prepare them for the neural network. This includes resizing all of our images to 64x64x64 pixels, normalizing the image data to center around 0 and scale to unit variance. This process will help us to keep consistency between images and downscale them for memory efficiency.

Tracking Data

We use *numpy* to normalize our patient age and keep track of statistics about the dataset. We keep track of how many images each patient has, store their age, and track patient IDs to help with our data splitting.

Output

images, ages_normalized, patient_ids, (age_mean, age_std)

Data Splitting Code

Packages

- *sklearn*: For splitting the data into Training and Test sets
- *numpy*: Array operations

Data Splitting

In this section we are going to split our data into a training set and test set. We need to make sure that the same patient images stay together. We don't want to pass a brain scan of a patient into our test set if we used that patient to train our model. This could allow our model to "cheat" and memorize that particular patient. We use a *sklearn* Class called GroupShuffleSplit which will randomly partition our data into an 80/20 split while maintaining groups. In this case it is maintaining each patient as a group! We then apply our split to the data and classifier (Scans / Ages) to form our training and test sets.

```
# Split the data using GroupShuffleSplit to prevent data leakage
splitter = GroupShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
train_idx, test_idx = next(splitter.split(images, ages_normalized, groups=patient_ids))

X_train, X_test = images[train_idx], images[test_idx]
y_train, y_test = ages_normalized[train_idx], ages_normalized[test_idx]

print(f"Training set size: {len(X_train)} (from {len(set(patient_ids[train_idx]))} patients)")
print(f"Test set size: {len(X_test)} (from {len(set(patient_ids[test_idx]))} patients)")
```

```
Training set size: 1092 (from 298 patients)
Test set size: 275 (from 75 patients)
```

BrainAgeDataset()

Packages

- *torch*: Main deep learning framework
- *torch.utils.data*: Base class for Datasets

Loading Data

Using *torch* we use *Dataset* as our base class for BrainAgeDataset. This will allow us easy and efficient access to our images and metadata for training. We initialize our class using our images and ages from our split sets of data.

Data Augmentations

You will see a couple of functions in this class called *random_noise*, *random_intensity*, and *random_flip.* We include these to add some slight augmentations to our images. They simulate real world possibilities such as scan noise, brightness variations between scanners, and brain symmetry. The hope here is that our model would be able to better handle more real world variations than what might exist in our dataset. This could help the model to better generalize to a wider variety of scans. The *clamp* function from *torch* helps to ensure only modest adjustments are made.

```python
class BrainAgeDataset(Dataset):
    def __init__(self, images, ages, is_train=True):
        self.images = torch.FloatTensor(images)
        self.ages = torch.FloatTensor(ages)
        self.is_train = is_train

    def random_noise(self, image, noise_factor=0.05):
        noise = torch.randn_like(image) * noise_factor
        return image + noise

    def random_intensity(self, image, factor_range=0.2):
        factor = 1.0 + torch.rand(1).item() * factor_range - factor_range/2
        return image * factor

    def random_flip(self, image):
        if torch.rand(1).item() > 0.5:
            return torch.flip(image, dims=[3])  # Flip along width
        return image

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = self.images[idx]
        age = self.ages[idx]

        if self.is_train:
            # Apply augmentations with probability
            if torch.rand(1).item() > 0.5:
                image = self.random_noise(image)
            if torch.rand(1).item() > 0.5:
                image = self.random_intensity(image)
            if torch.rand(1).item() > 0.5:
                image = self.random_flip(image)

            # Ensure values are in reasonable range
            image = torch.clamp(image, -3, 3)

        return image, age
```

ResBlock()

Packages

- *torch.nn*: Neural Network modules

What it does

ResBlock allows us to ensure that information can flow through our network layers without getting distorted by creating shortcuts or bypasses where possible.

Creating our Residual Network

Using *torch* we use *nn.Module* as our base class for ResBlock.

- *Conv3d:* 3D convolution for processing our brain scans
- *BatchNorm3d:* Normalized data for more stable training
- *Dropout3d:* Randomly turns off features to protect against overtraining
- *ReLU:* Activation Function

downsample is used to adjust dimensions if needed and allows information to bypass main path.

forward() allows us to follow steps in order by providing a precise “blueprint” that our data must follow. All *nn.Module* must have a *forward* function.

```python
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels, dropout=0.0):
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv3d(in_channels, out_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm3d(out_channels)
        self.conv2 = nn.Conv3d(out_channels, out_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm3d(out_channels)
        self.dropout = nn.Dropout3d(dropout) if dropout > 0 else None

        self.downsample = None
        if in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv3d(in_channels, out_channels, kernel_size=1),
                nn.BatchNorm3d(out_channels)
            )
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        if self.dropout is not None:
            out = self.dropout(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
        return out
```

BrainAgeCNN()

Packages

- *torch.nn*: Neural Network modules
  - *nn.Conv3d*: 3D convolutions
  - *nn.BatchNorm3d:* normalization
  - *nn.ReLU:* activation functions
  - *nn.Dropout3d:* regularization
  - *nn.Linear:* fully connected layers
  - *nn.Sequential* : layer organization

Initial Block

Here we take in the 3D brain scans from our *BrainAgeDataset* and process them using 3D convolution. In this stage we also apply some dropout to prevent overfitting.

Residual Blocks

We leverage our *ResBlock* class in 3 stages increasing the number of channels each time. This allows us to find general trends in the brain scans and then look for more specific patterns as we go deeper into the net. We also leverage an increase in dropout as we increase the number of channels.

Hardware Specification

In this class, we attempt to figure out which hardware is available to the program. For NVIDA systems we leverage CUDA. For Apple Silicon we use MPS which doesn't have support for *MaxPool3d* so I opted for *Conv3d.* For these systems. In all other cases we default to using the CPU, however this will increase run times.

Processing

This class averaged all of the spatial information and runs through 3 fully connected layers. It gradually reduces the information at each layer (128->64->32->1) to make a final aged prediction as single normalized value. We also leverage dropout heavily to help prevent overfitting as the network gets deeper.

What does it mean?

We can look at this class in the following way. Take an image of a brain, process it though increasingly complex filters, learn patterns associated with the brain aging, and make a guess about the age of the brain. While we do this, we have many safety features that aim to prevent us from memorizing our training data.

evaluate_metrics()

Packages

- *numpy*: Numerical computations

Evaluation

This code takes in our true and predicted values and uses them to create a more readable expression of results. If you remember, we normalized our ages during the data loading phases. This makes results a little bit more difficult to understand than something more tangible. In this case we denormalize the values and generate a response in true years.

Output

This code returns the following metrics

- MAE (Mean Absolute Error): This is the average absolute difference between our prediction and the true value.
- MSE (Mean Squared Error): This is the average of square differences between the values.
- RMSE (Root Mean Squared Error): This is the root square of the MSE which gives a better sense of error magnitude.

```python
def evaluate_metrics(y_true, y_pred, age_mean, age_std):
    # Denormalize predictions and true values
    y_true_denorm = y_true * age_std + age_mean
    y_pred_denorm = y_pred * age_std + age_mean

    # Calculate regression metrics
    mae = np.mean(np.abs(y_true_denorm - y_pred_denorm))
    mse = np.mean((y_true_denorm - y_pred_denorm) ** 2)
    rmse = np.sqrt(mse)

    return {
        'mae': mae,
        'mse': mse,
        'rmse': rmse
    }
```

train_model()

Packages

- *torch.optim:* Optimization algorithms (AdamW optimizer)
- *torch.nn:* Neural network components
- *torch.cuda/mps:* Different GPU supports
- *torch.data.utils:* Data loading utilities
- *numpy:* Numerical and array operations
- *matplotlib:* Visualization of metrics

Training Loop

We run this loop up to 50 times (you may adjust the number of epochs). There is a 3 epoch warmup to help the learning rate gradually reach full capacity. For each epoch we train on a batch of brain scans, validate the performance, track metics, adjust learning rate, save the best model, and check for early stopping conditions.

Visualizations

Throughout the training, we keep track of metrics which will be analyzed in the results section. We leverage *matplotlib* to generate these plots.

Key Features

- We leverage several training techniques such as learning rate warmup, gradient clipping, early stopping, learning rate scheduling and regularization.
- We support different hardware options which has been discussed previously as well as the tradeoffs.
- We provide metrics after each epoch and also track metrics for data visualizations. These visualizations are discussed and analyzed in the next section.

What does it mean?

This function is quite long so I have left many of the technical details in the code itself. The is a training implementation that outputs a trained model, performance metrics, and saves the best model as an output. We have implemented several pieces of functionality that provide error handling, performance monitoring, and optimization to help train our deep neural network.

Running the Model

It may take a considerable amount of time to run the model. Depending on your system it could take several hours to days. I have included a sample output log as well as a complete model as a .pth that you may use if you are unable to run this due to technical limitations of your system

Running the Model

Output

You will begin to see output like what is shown here. For my system, it took around 10 minutes for the first epoch to show results. As for complete training, upwards of 5-6 hours on average. If you wish to run through a complete training, please be patient. If not, you can skip to the results section here where I provide output and analysis using two different learning rates.

CUDA / NVIDIA

Hopefully someone is able to run this on a CUDA / NVIDIA system. It is my understanding that you will experience considerably faster training times.

Apple Silicon MPS

While I expected a speedup here, some of the faster torch classes are not support yet for MPS so results were not much faster.

CPU

Goodluck. It is going to take a considerable amount of time. If you have a relatively modern processor it may still be reasonable to train.

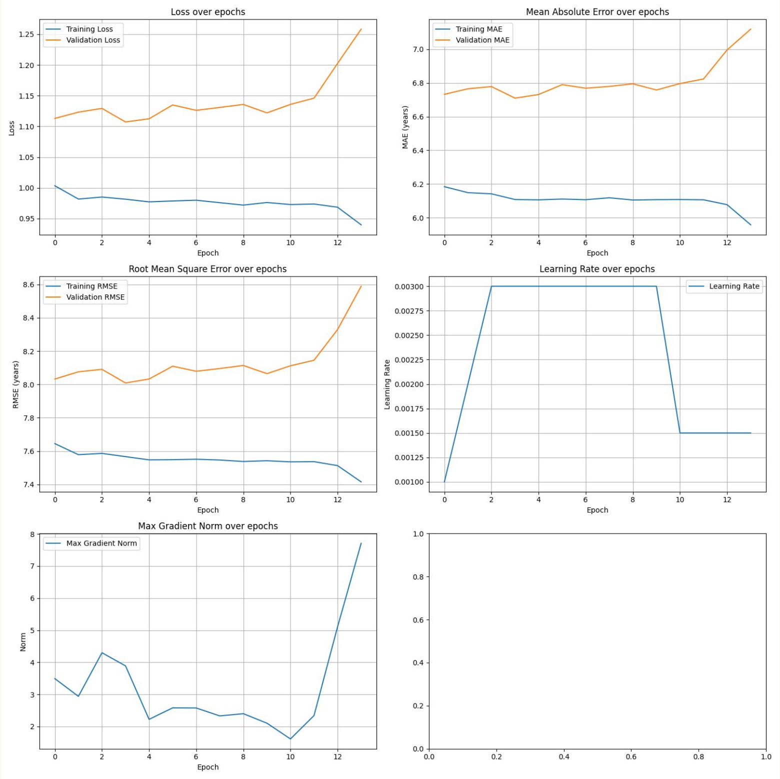
```
Using device: mps
Note: Using MPS (Apple Silicon GPU)
Using strided convolution for MPS device
Starting training...

Epoch [1/50]
Learning Rate: 0.001000
Max Gradient Norm: 3.4927
Training Loss: 1.0034
Validation Loss: 1.1130

Training Metrics:
MAE: 6.18 years
RMSE: 7.65 years

Validation Metrics:
MAE: 6.73 years
RMSE: 8.03 years
--------------------------------------------------

Epoch [2/50]
Learning Rate: 0.002000
Max Gradient Norm: 2.9394
Training Loss: 0.9818
...
--------------------------------------------------

Early stopping triggered at epoch 13
Best validation loss was 1.1072 at epoch 3
```

Reviewing Results

Learning Rate .003 Data

Epoch	Learning Rate	Max Gradient Norm	Training Loss	Validation Loss	Training MAE	Training RMSE	Validation MAE	Validation RMSE
1	0.001	3.4927	1.0034	1.113	6.18	7.65	6.73	8.03
2	0.002	2.9394	0.9818	1.1231	6.15	7.58	6.77	8.08
3	0.003	4.2989	0.9851	1.1293	6.14	7.59	6.78	8.09
4	0.003	3.8886	0.9816	1.1072	6.11	7.57	6.71	8.01
5	0.003	2.2234	0.9772	1.1124	6.11	7.55	6.73	8.03
6	0.003	2.5815	0.9787	1.1349	6.11	7.55	6.79	8.11
7	0.003	2.5775	0.9799	1.1261	6.11	7.55	6.77	8.08
8	0.003	2.3303	0.9759	1.1308	6.12	7.55	6.78	8.1
9	0.003	2.3996	0.972	1.1357	6.1	7.54	6.8	8.11
10	0.0015	2.1045	0.9762	1.122	6.11	7.54	6.76	8.06
11	0.0015	1.6124	0.9729	1.1358	6.11	7.54	6.8	8.11
12	0.0015	2.3416	0.9737	1.1459	6.11	7.54	6.82	8.15
13	0.0015	5.1082	0.9685	1.2025	6.08	7.51	7.0	8.33

Learning Rate .003 Charts



Learning Rate .003 Analysis

Loss Curves

We see a gradual decrease in training loss of epochs which is good. Validation loss is consistently higher than training loss and explodes after epoch 10. This is suggestive of overfitting.

Mean Absolute Error

Our training MAE is relatively stable and decreasing while our validation MAE begins to explode after epoch 10. This is suggestive of poor generalization.

Root Mean Square Error

RMSE shows a similar pattern to MAE and further suggests that our model is overfit.

Learning Rate

We see stability after warm up and see that scheduling then tries to adjust learning rate. After this adjustment, our model explodes.

Max Gradient Norm

The chart shows moderate fluctuations until epoch 10 where it explodes. This is consistent with the deterioration that we see in our error metrics.

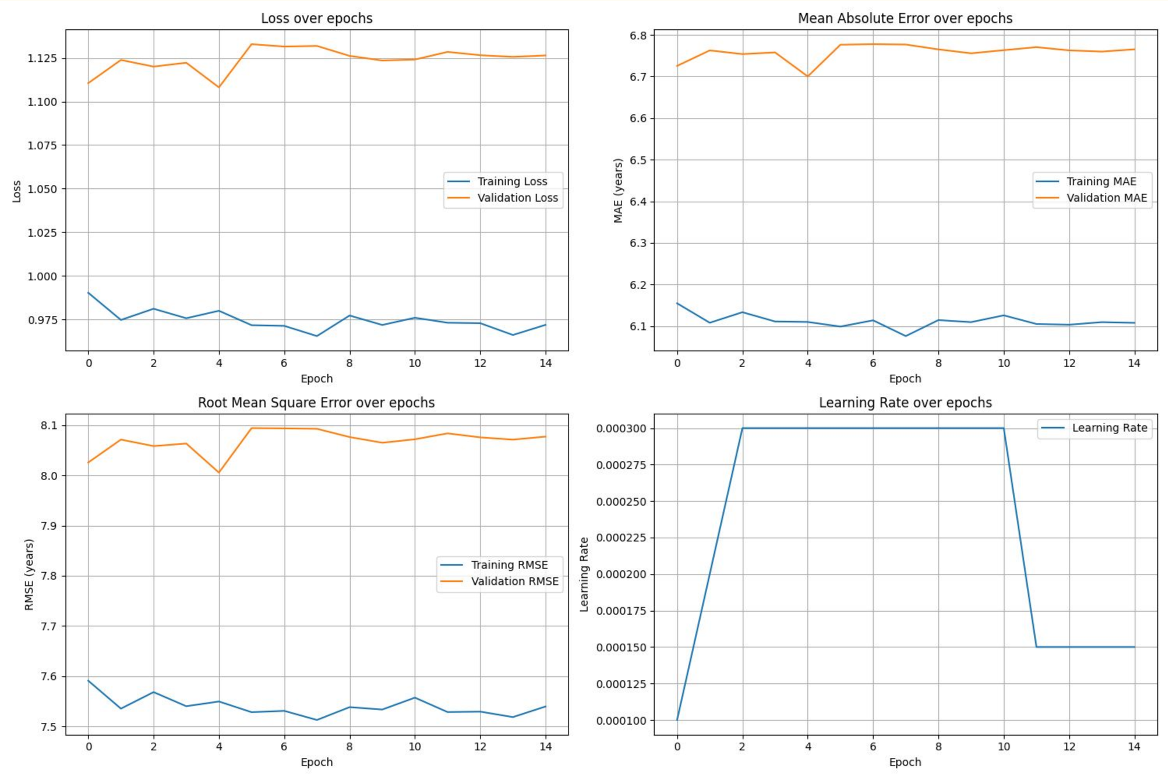
Results

We see strong indications that our model is overfit and is generalizing poorly when viewing unseen data. It appears that our attempt to adjust learning rate has caused the model to overfit and explode. Our best epoch shows a performance of ~6.8 years which is actually pretty good. In the next iteration we are going to try to run the training with a less aggressive initial learning rate. We hope that this adjustment to learning rate will show more stability in the model.

Learning Rate .0003 Data

Epoch	Learning Rate	Max Gradient Norm	Training Loss	Validation Loss	Training MAE	Training RMSE	Validation MAE	Validation RMSE
1	0.0001	2.8487	0.9904	1.1105	6.15	7.59	6.73	8.03
2	0.0002	3.102	0.9748	1.1238	6.11	7.54	6.76	8.07
3	0.0003	3.4633	0.9812	1.12	6.13	7.57	6.75	8.06
4	0.0003	2.7699	0.9758	1.1222	6.11	7.54	6.76	8.06
5	0.0003	2.75	0.98	1.1081	6.11	7.55	6.7	8.01
6	0.0003	3.1469	0.9718	1.1328	6.1	7.53	6.78	8.09
7	0.0003	2.8915	0.9714	1.1315	6.11	7.53	6.78	8.09
8	0.0003	2.6939	0.9655	1.1318	6.08	7.51	6.78	8.09
9	0.0003	2.6234	0.9773	1.1261	6.11	7.54	6.77	8.08
10	0.0003	2.6274	0.9719	1.1235	6.11	7.53	6.76	8.06
11	0.00015	2.3751	0.976	1.124	6.13	7.56	6.76	8.07
12	0.00015	1.9978	0.9732	1.1284	6.1	7.53	6.77	8.08
13	0.00015	2.3558	0.9729	1.1265	6.1	7.53	6.76	8.08
14	0.00015	2.5485	0.9662	1.1256	6.11	7.52	6.76	8.07

Learning Rate .0003 Charts



Step size .0003 Analysis

Loss Curves

Much more stability show consistent loss with gradual improvement. We also see a smaller gap between training and validation loss. We would like to see better generalization in our validation loss but this is pretty good.

Mean Absolute Error

Similar to our previous statement. We are showing much more stability with no explosion (~6.75 years in our Validation set).

Root Mean Square Error

RMSE shows a similar pattern to MAE (~8.06 years). This suggests that we have some guesses that are a bit higher than our absolute average. This is still an acceptable gap.

Learning Rate

We see stability after warm up and see that scheduling then tries to adjust learning rate. After this adjustment, our model does not explode. This suggest a reasonable generalization.

Max Gradient Norm

No explosion.

Results

We can see that controlling our learning rate to be less aggressive has resulted in more stable results. Our model performs best at epoch 4 which is a little early. It is possibly suggestive that our model is just set up pretty well and learns quickly. It is also possible that our algorithm is stuck at a local min and could be performing better. We could adjust our warm up period, test with different learning rate scheduling methods, or expand our early stopping period to see if further improvement is possible. We may still have concerns over potential overfitting as we see that our gaps between test and validation metrics are fairly consistent. One thing we could try to improve this overfitting issue is to adjust our data augmentation methods to be more aggressive. In my opinion, I think that a larger dataset with more images would help this model to better generalize.

Closing Thoughts and Sources

Resources

Cursor:

https://www.cursor.com/en

I leveraged cursor in this project to help with code quality improvements, and improving the look and feel of comments / markdown. This is an extremely powerful tool. If you are using AI assistants to help with programming tasks be sure to always review and test results independently. I experienced a lot of hallucinations throughout this process.

Udemy:

https://www.udemy.com/course/deeplearning/?couponCode=ST11MT170325G3

While this course does cost money (fortunately I have subscription through work) I would highly recommend it. The CNN intuition section provides really great details and a solid conceptual understanding of CNNs.

Academic Paper:

https://www.nature.com/articles/s41598-023-49514-2

I used this paper as an inspiration for this self learning tutorial. It provides a much better implementation than I was able to create using just the power of my machine. Their results are top notch yielding classifications within 3-4 years over a wide age audience. If this topic interests you I highly recommend checking out this paper.

Time:

This took an immense amount of time (120 hours +). I hope to use this topic in the final project for this course. If anyone has any questions or would like to work together in a group for the final, please make a post on Ed and we can connect.