

Me:

There are 81 total points, and nearly 30 fall in the 4 interval class. how do i prevent my model from predicting 4 every time

ChatGPT:

When nearly 30 out of 81 total data points fall into the same class (e.g., `IntervalClass = 4`), your model is prone to **class imbalance**, leading it to predict 4 disproportionately often because it minimizes the overall loss by favoring the majority class. To address this, consider the following strategies:

### Option 1: Oversample Minority Classes

Duplicate samples from underrepresented classes so all classes have roughly equal representation.

```
python
from sklearn.utils import resample # Separate majority and minority classes
majority_class = train_data[train_data['IntervalClass'] == 4]
minority_classes = train_data[train_data['IntervalClass'] != 4] # Oversample the minority classes
minority_oversampled = resample(minority_classes, replace=True,
                                n_samples=len(majority_class), random_state=42) # Combine back into a balanced dataset
balanced_train_data = pd.concat([majority_class, minority_oversampled])
```

Me:

are there other methods to reduce overfitting besides dropout for this net = nn.Sequential(nn.Linear(all\_features.shape[1], 128), nn.ReLU(), nn.Dropout(p=0.3), nn.Linear(128, 64), nn.ReLU(), nn.Dropout(p=0.3), nn.Linear(64, 24) )

ChatGPT:

Yes, there are **several other methods** to reduce overfitting besides **Dropout** for your network. Here are the most effective strategies:

---

## 1. L2 Regularization (Weight Decay)

- Penalizes large weights by adding their square (L2 norm) to the loss function.
- Helps prevent the model from fitting noise in the data.
- PyTorch provides this via the `weight_decay` parameter in optimizers.

### How to Use:

```
python
Copy code
```

```
optimizer = torch.optim.Adam(net.parameters(), lr=0.001,  
weight_decay=1e-4)
```

- `weight_decay=1e-4` is a common starting value.
- 

## 2. Early Stopping

- Monitor validation loss during training and stop once it stops improving.
  - You already implemented this, but it's important to emphasize its role.
- 

## 3. Batch Normalization

- Adds normalization to the output of each layer to stabilize training.
- Reduces internal covariate shift and acts as a form of regularization.
- Place `nn.BatchNorm1d` after each fully connected layer.

### Modified Network:

python

Copy code

```
net = nn.Sequential(  
    nn.Linear(all_features.shape[1], 128),  
    nn.BatchNorm1d(128), # Batch normalization  
    nn.ReLU(),  
    nn.Linear(128, 64),  
    nn.BatchNorm1d(64),  
    nn.ReLU(),  
    nn.Linear(64, 24)  
)
```

---

## 4. Data Augmentation

- If you are dealing with image data or any sequential input, augmenting the data artificially helps reduce overfitting.
- Common augmentations: flipping, rotation, scaling, noise addition, or random erasing.

### For Numerical Features:

- Add small Gaussian noise to the input features during training to increase generalization.

python

Copy code

```
noisy_input = input_features + torch.randn_like(input_features) * 0.01
# Gaussian noise
```

---

## 5. Reduce Network Complexity

- Overly complex networks can overfit small datasets. Use smaller hidden layer sizes or fewer layers.

### Example:

python

Copy code

```
net = nn.Sequential(
    nn.Linear(all_features.shape[1], 64), # Reduce size
    nn.ReLU(),
    nn.Linear(64, 32),
    nn.ReLU(),
    nn.Linear(32, 24)
)
```

---

## 6. Gradient Clipping

- Prevents exploding gradients, which may indirectly help reduce overfitting.
- Set a threshold for gradient norms during backpropagation.

### How to Use:

python

Copy code

```
torch.nn.utils.clip_grad_norm_(net.parameters(), max_norm=1.0)
```

---

## 7. Label Smoothing

- Slightly "softens" the true labels to prevent the model from becoming overconfident.
- Helps regularize the training process.

**Example** (CrossEntropy Loss with Label Smoothing in PyTorch):

python

Copy code

```
loss_fn = nn.CrossEntropyLoss(label_smoothing=0.1)
```

---

## 8. Use Learning Rate Scheduling

- Reduce the learning rate when validation loss stops improving.
- Use `ReduceLROnPlateau` scheduler.

**How to Use:**

python

Copy code

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,  
'min', factor=0.5, patience=5)
```

Call this after each epoch:

python

Copy code

```
scheduler.step(val_loss)
```

---

## 9. Train on More Data

- If possible, gather more training data to improve generalization.
- 

## 10. Ensemble Learning

- Combine predictions from multiple models (e.g., bagging, boosting, or voting) to reduce overfitting.
  - Train models on slightly different data splits or initializations.
-

## Combined Example:

Here's an updated version of your network with multiple methods applied:

python

Copy code

```
net = nn.Sequential(
    nn.Linear(all_features.shape[1], 128),
    nn.BatchNorm1d(128), # Batch Normalization
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.BatchNorm1d(64),
    nn.ReLU(),
    nn.Linear(64, 24)
)

# Optimizer with weight decay
optimizer = torch.optim.Adam(net.parameters(), lr=0.001,
weight_decay=1e-4)

# Loss function with label smoothing
loss_fn = nn.CrossEntropyLoss(label_smoothing=0.1)
```

---

## Summary of Key Techniques:

### 1. Weight Decay (L2 Regularization)

Me:

output names of all marathons

ChatGPT:

To extract and output the **names of all marathons** from your dataset, we need to focus on the "Activity Name" column for activities that match the marathon criteria.

The following function will filter and print the marathon names based on the `marathon_list` keyword pattern:

---

## Code to Extract Marathon Names

Add this function to your script and call it in the `main` function:

python

Copy code

```
import re
```

```
def extract_marathon_names(marathon_training_list):
    # Compile a pattern for marathons
    marathon_pattern = '|'.join(map(re.escape, marathon_list))

    # Extract marathon names
    marathon_names = []
    for (name, result_dfs) in marathon_training_list:
        for df in result_dfs:
            # Filter activities that match the marathon list
            marathon_activities = df[df["Activity
Name"].str.contains(marathon_pattern, case=False, na=False)]
            marathon_names.extend(marathon_activities["Activity
Name"].tolist())

    # Print all marathon names
    print("Marathon Names:")
    for i, marathon_name in enumerate(set(marathon_names), start=1):
        print(f"{i}. {marathon_name}")
```

---

## Update `main` to Call the Function

Call the new function after processing the data:

python

Copy code

```
def main():
    marathon_training_list = split_blocks(get_data())
    run_ct_rest_days = count_run_ct_rest_days(marathon_training_list)
    weekly_values =
calculate_weekly_distance_time(marathon_training_list)
```

```
    long_run_data = find_long_run(marathon_training_list)
    weather_condition_data =
scrape_weather_condition(marathon_training_list)
    marathon_df = create_dataframe(marathon_training_list,
run_ct_rest_days, weekly_values, long_run_data,
weather_condition_data)
    marathon_df.to_csv('marathon_training_data.csv', index=False)

# Output marathon names
extract_marathon_names(marathon_training_list)

if __name__ == "__main__":
    main()
```

---

## Explanation

1. **Pattern Matching:** The `marathon_list` contains keywords like "Marathon", "Maraton", etc. We create a regex pattern to search for these words in the "Activity Name" column.
  2. **Filter Activities:** Activities whose names match the marathon keywords are selected.
  3. **Collect Unique Names:** The `set` function ensures that only unique marathon names are printed.
  4. **Output:** Prints a numbered list of marathon names.
- 

## Sample Output

If the dataset contains marathons like:

- "Boston Marathon 2023"
- "New York City Marathon"
- "CIM 2022"

The output will be:

markdown

Copy code

Marathon Names:

1. Boston Marathon 2023
2. New York City Marathon

### 3. CIM 2022