Neural Architecture Search

Introduction

Neural Architecture Search (NAS) is a technique that helps in automatically creating neural network designs. It is basically a technique for automating the design of Neural Network models pertaining to the type of dataset that is going to be consumed. NAS aims to find the best network structure for tasks like image recognition or language processing by exploring various design options. By automating this process, NAS improves the performance of neural networks and reduces time and effort compared to manually designing and tuning these networks.

Neural Architecture Search Implementation:-

Neural Architecture Search(NAS) implementation overall can be summarized under three modules :-

- Search Space
- Search Strategy
- Performance Estimation Strategy

Search Space

A Search space consists of all the parameters being considered into the design of our optimal Neural Network model. The choices of parameters can be outlined as many things such as :- number of layers, whether a layer needs to be considered(Example:- an addition convolutional hidden layer, a dense hidden layer etc.,), kernel size of respective convolutional layers, number of hidden units in dense layers, dropout choice and many more. These choices help the NAS implementation to optimize the NN model design to find the best model suitable for our demands.

Search Strategy

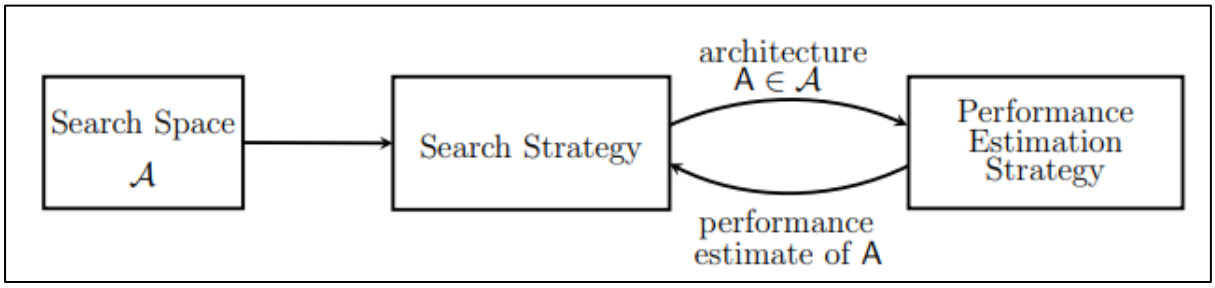
A Search Strategy works inside the search space implementing a particular strategy to explore (all or some) the mentioned parameter choices. Different strategies offer various trade-offs between exploration efficiency and computational cost benefits. Here are some prominent search strategies used in NAS :-

- **Random Search** :- Random Search is the simplest NAS strategy. It randomly samples architectures from the search space and evaluates them.
- **Grid Search** :- Grid Search systematically explored by evaluating all possible combinations of hyperparameters.
- **Bayesian Optimization** :- This optimization search builds a probabilistic model of the objective function and used it to select the most promising architectures to evaluate next.
- **Reinforcement Learning**:- RL-based controller generates architectures and works to find the most optimum architecture using reward signals.

In this demo, we implemented the baseline search strategy which is “Random Search” strategy.

Performance Estimation Strategy

In Neural Architecture Search (NAS), performance estimation strategies are crucial for evaluating the potential of candidate architectures. These strategies aim to provide a quick and reasonably accurate estimate of how well a given architecture might perform. This strategy is thus important as it also guides the search process. The simplest way of performing estimation of an architecture’s performance is to train it on training data and evaluate its performance on validation data.



NAS Implementation *(Source: arXiv:1808.05377)*

Process of Neural Architecture Search

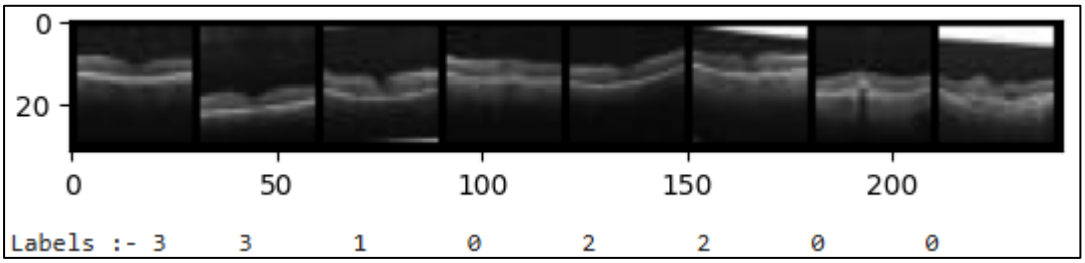
Thus, with defining the three important modules of NAS, we get a decent estimate of what consists a NAS implementation. Now we define the process of working of NAS in steps as follows :-

- **Prepare the dataset**
  The dataset we are considering in this code demo is octmnist images dataset from MedMNIST. *The OCTMNIST dataset is a series of benchmark datasets for medical image analysis. OCTMNIST specifically comprises Optical Coherence Tomography (OCT) images for retinal disease classification. [To access this dataset, run :- `pip install medmnist`]*

  The dataset is divide into two sets :- training set to train the candidate architecture, validation set to evaluate candidate architecture performance. [70% training proportion : 30% validaton proportion]

```
Training set: (76516, 1, 28, 28), (76516, 1)
Validation set: (32793, 1, 28, 28), (32793, 1)
```

  The octmnist dataset contains of images that can be classified into four classes :- CNV(Chloroidal Neovascularization), DME(Diabetic Macular Edema), Drusen, Normal.



- **Define a Basic Artificial Neural Network Model**

```python
# Basic CNN model
class CNN(nn.Module):
    def __init__(self, conv_layers, numhidden, input_dim, dropout):
        super(CNN, self).__init__()
        layers = []
        input_channels=1
        for i in range(conv_layers):
            layers.append(nn.Conv2d(input_channels, 32, kernel_size=3))
            layers.append(nn.ReLU())
            layers.append(nn.MaxPool2d(2))
            input_channels=32
            input_dim -= 2
            input_dim /= 2
            input_dim = int(input_dim)
            if(input_dim <= 2): break
        if(dropout > 0): layers.append(nn.Dropout(p=dropout))
        self.conv = nn.Sequential(*layers)
        self.fc1 = nn.Linear(32*input_dim*input_dim, numhidden)
        self.fc2 = nn.Linear(numhidden, 4)

    def forward(self, x):
        x = self.conv(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

This is a basic CNN model we built with certain parameters consisting the search space.

- **Define a Search Space** (based on the parameters pertaining to the basic model)

```python
# Search space -- denotes the choices of parameters the basic model can take
search_space = {
    'conv_layers' : (1, 5),
    'numhidden' : (16, 512),
    'dropout' : (0, 0.2, 0.5, 0.8)
}
```

This basic search space defined the number of additional convolutional layers that can be considered(between 1 and 5), the number of hidden units to consider in penultimate dense layer, and the parameter for dropout layer. NOTE: The search space is recommended to be configured with a yaml file.

- **Define a Performance Estimation Strategy (Model Evaluator)**

```python
# Model Evaluator
def evaluate_model(model, epochs=3):
    optimizer = optim.Adam(model.parameters())
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            data = data.to(device)
            target = target.view(-1)
            target = target.to(target)

            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    # Validate the model
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in val_loader:
            data = data.to(device)
            target = target.view(-1)
            target = target.to(target)

            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    accuracy = correct / len(val_loader.dataset)
    return accuracy
```

This is a simple model evaluation strategy which trains a candidate model on a training dataset and evaluates its validation accuracy with a validation dataset.

- **Decide on a Search Strategy**

```python
for _ in range(num_trials):
    conv_layers = torch.randint(search_space['conv_layers'][0], search_space['conv_layers'][-1], (1,)).item()
    numhidden = torch.randint(search_space['numhidden'][0], search_space['numhidden'][-1], (1,)).item()
    dropout = search_space['dropout'][torch.randint(0, 3, (1,))]
```

We have implemented Random Search Strategy to explore the search space.

- **Implement Neural Architecture Search** by searching through the search space generating candidate architectures and estimating them using the model evaluator.

```
# Searching algorithm -- implements random search strategy
def search(num_trials, search_space, input_dim):
    results = []
    for _ in range(num_trials):
        conv_layers = torch.randint(search_space['conv_layers'][0], search_space['conv_layers'][-1], (1,)).item()
        numhidden = torch.randint(search_space['numhidden'][0], search_space['numhidden'][-1], (1,)).item()
        dropout = search_space['dropout'][torch.randint(0, 4, (1,))]

        cnn = CNN(conv_layers, numhidden, input_dim, dropout)
        cnn.to(device)
        accuracy = evaluate_model(cnn)
        results.append((conv_layers, numhidden, dropout, accuracy))

    return results
```

- **Compare the candidate architectures and retrieve the best parameters for model.**

```
Conv layers: 2, Hidden units: 219, Dropout: 0.2, Accuracy: 0.8193
Conv layers: 2, Hidden units: 261, Dropout: 0.2, Accuracy: 0.8089
Conv layers: 4, Hidden units: 264, Dropout: 0.2, Accuracy: 0.7949
Conv layers: 1, Hidden units: 98, Dropout: 0.8, Accuracy: 0.7679
Conv layers: 2, Hidden units: 34, Dropout: 0, Accuracy: 0.8043
Conv layers: 2, Hidden units: 31, Dropout: 0.8, Accuracy: 0.7726
Conv layers: 4, Hidden units: 357, Dropout: 0.8, Accuracy: 0.3413
Conv layers: 2, Hidden units: 377, Dropout: 0.2, Accuracy: 0.8097
Conv layers: 3, Hidden units: 272, Dropout: 0.5, Accuracy: 0.8018
Conv layers: 4, Hidden units: 435, Dropout: 0.8, Accuracy: 0.3413
Conv layers: 4, Hidden units: 431, Dropout: 0.5, Accuracy: 0.8074
Conv layers: 1, Hidden units: 310, Dropout: 0, Accuracy: 0.8002
Conv layers: 2, Hidden units: 475, Dropout: 0.8, Accuracy: 0.7930
Conv layers: 2, Hidden units: 85, Dropout: 0.5, Accuracy: 0.7925
Conv layers: 1, Hidden units: 237, Dropout: 0.5, Accuracy: 0.7985
```

References:-

arXiv:1808.05377

NNI Documentation — Neural Network Intelligence

Overview of Neural Architecture Search | Paperspace Blog

xiaoiker/NAS-With-Code: Neural Architecture Search (NAS) papers with code (github.com)

arXiv:1611.02167

Efficient Neural Architecture Search via Parameters Sharing (mlr.press)