Supplementary Material for LLMatic: Neural Architecture Search via Large Language Models and Quality-Diversity Optimization

Pseudocodes

Algorithm 1: Mutation Operator

```
input  : selected_network, selected_prompt, selected_temperature;
prompt = selected_network + selected_prompt;
generated_network = LLM_generation(prompt, selected_temperature);
return : generated network;
```

Algorithm 2: Crossover Operator

```
input  : selected_network_1, selected_network_2
prompt = selected_network_1 + selected_network_2 + crossover_prompt
generated_network = LLM_generation(prompt, crossover_temperature)
return : generated network;
```

Algorithm 3: Temperature Mutation

```
input  : selected_temperature, network_loss, best_loss;
if network_accuracy >= best_accuracy then
    temperature = starting_temperature + 0.05;
else
    temperature = starting_temperature - 0.05;
end
return : temperature;
```

Algorithm 4: Addition in archive

```
input : all_generated_networks_and_prompts, best_loss;
all_networks, all_prompts, temperatures = all_generated_networks_and_prompts;
foreach i in length(all_networks) do
    FLOPS = calculate_flops(all_networks[i]);
    depth_to_width_ratio = calculate_depth_to_width_ratio(all_networks[i]l);
    normalized_score = normalize(FLOPS, depth_to_width_ratio);
    find_nearest_centroid(normalized_score);
    if cell is free in network_archive then
        add network;
    else
        if cell is filled and accuracy of network is more then
            add network;
        else
            Don't add the network;
        end
    end
    if accuracy(all_networks[i]) >= best_accuracy then
        network_fitness_check.append(True);
    else
        network_fitness_check.append(False);
    end
    prompt = to_integer(all_prompts[i]l);
    temperature = temperatures[i];
    normalized_score = normalize(prompt, temperature);
    find_nearest_centroid(normalized_score);
    if network_fitness_check[i] == True then
        collective_fitness_score[prompt] + = 1;
    else
        pass;
    end
    if cell is free in prompt_archive then
        add prompt individual;
    else
        add prompt individual if collective fitness score of the prompt is greater;
    end
end
```

Generated Networks

Generated network 1:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 256, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(512, 1024, kernel_size=3, padding=1)
        self.conv6 = nn.Conv2d(1024, 1024, kernel_size=3, padding=1)
        self.conv7 = nn.Conv2d(1024, 2048, kernel_size=3, padding=1)
        self.conv8 = nn.Conv2d(2048, 2048, kernel_size=3, padding=1)

        self.bn1 = nn.BatchNorm2d(256)
        self.bn2 = nn.BatchNorm2d(512)
        self.bn3 = nn.BatchNorm2d(1024)
        self.bn4 = nn.BatchNorm2d(2048)

        self.pool = nn.MaxPool2d(2, 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))

        self.fc = nn.Linear(2048, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(F.relu(self.bn1(self.conv2(x))))
        x = F.relu(self.bn2(self.conv3(x)))
        x = self.pool(F.relu(self.bn2(self.conv4(x))))
        x = F.relu(self.bn3(self.conv5(x)))
        x = self.pool(F.relu(self.bn3(self.conv6(x))))
        x = F.relu(self.bn4(self.conv7(x)))
        x = self.pool(F.relu(self.bn4(self.conv8(x))))

        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

Generated network 2:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
```

```
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2, 2)
        self.adaptive_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(128, 512)
        self.fc2 = nn.Linear(512, 10)
        self.residual_conv1 = nn.Conv2d(16, 64, 1, stride=4, padding
            =0)  # 1x1 convolution for matching dimensions
        self.residual_conv2 = nn.Conv2d(64, 128, 1, stride=2,
            padding=0)  # 1x1 convolution for matching dimensions

    def forward(self, x):
        x1 = self.pool(F.relu(self.bn1(self.conv1(x))))
        x2 = self.pool(F.relu(self.bn2(self.conv2(x1))))
        x_residual1 = self.residual_conv1(x1)  # Apply 1x1
            convolution to match dimensions
        x3 = self.pool(F.relu(self.bn3(self.conv3(x2))))
        x3 += x_residual1  # Add the residual connection
        x4 = self.pool(F.relu(self.bn4(self.conv4(x3))))
        x_residual2 = self.residual_conv2(x3)  # Apply 1x1
            convolution to match dimensions
        x4 += x_residual2  # Add the residual connection
        x = self.adaptive_pool(x4)
        x = x.view(-1, 128)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Generated network 3:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 1024, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(1024, 1024, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(1024, 2048, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(1024)
        self.bn2 = nn.BatchNorm2d(2048)

        self.pool = nn.MaxPool2d(2, 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))

        self.fc = nn.Linear(2048, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(F.relu(self.bn1(self.conv2(x))))
        x = self.pool(F.relu(self.bn2(self.conv3(x))))
```

```
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)  # flatten all dimensions except the
             batch dimension
        x = self.fc(x)

        return x
```

Generated network 4:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.25)

        self.lstm_input_size = 128 * 16 * 16
        self.lstm_hidden_size = 256
        self.lstm_num_layers = 2

        self.lstm = nn.LSTM(input_size=self.lstm_input_size,
                            hidden_size=self.lstm_hidden_size,
                            num_layers=self.lstm_num_layers,
                            batch_first=True)

        self.fc1 = nn.Linear(self.lstm_hidden_size, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.dropout(x)

        x = x.view(x.size(0), 1, -1)  # Reshape tensor for LSTM
            input
        x, _ = self.lstm(x)  # LSTM output
        x = x[:, -1, :]  # Last LSTM output

        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Generated network 5:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
```

```
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.dropout1 = nn.Dropout2d(0.5)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.dropout2 = nn.Dropout2d(0.5)

        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm2d(256)
        self.dropout3 = nn.Dropout2d(0.5)

        self.fc1 = nn.Linear(256*4*4, 10)

    def forward(self, x):
        out1 = F.relu(self.bn1(self.conv1(x)))
        out2 = self.dropout1(F.relu(self.bn2(self.conv2(out1))))
        out2 += out1 # skip connection
        out2 = F.max_pool2d(out2, 2)

        out3 = F.relu(self.bn3(self.conv3(out2)))
        out4 = self.dropout2(F.relu(self.bn4(self.conv4(out3))))
        out4 += out3 # skip connection
        out4 = F.max_pool2d(out4, 2)

        out5 = F.relu(self.bn5(self.conv5(out4)))
        out6 = self.dropout3(F.relu(self.bn6(self.conv6(out5))))
        out6 += out5 # skip connection
        out6 = F.max_pool2d(out6, 2)

        out6 = out6.view(out6.size(0), -1) # flatten the tensor
        out7 = self.fc1(out6)

        return out7
```