lab05

코드에 대한 설명은 코드에 안의 주석으로 설명해놨습니다.

약간의 쉬운 내용이나 중요도가 낮은 내용들은 코드가 아예 없거나 주석이 없습니다.

훈련 데이터 업데이트 및 손실 값 계산

```
for (int c = 0; c < 3; c++)
{
  // 손실값 초기화
  double Loss = 0;
  // 학습률 초기화
  double lr = 0.02;
  // True Positive 수 초기화
  int nTP = 0;

  for (int i = n_Index; i < n_Index + m_nBatch; ++i) {
    // 예측값 배열 초기화
    double y_perceptron[10];
    for (int k = 0; k < 10; k++)
    {
      double y = 0;

      // 분류기 1: 28*28의 모든 화소를 입력을 받음
      if (c == 0)
      {
        for (int x = 0; x < 28 * 28; ++x)
          y += m_MnistTrainInput[i][x] * m_W[c][k][x];
      }
      else if (c == 1)
      {
        // 분류기 2:
        // 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)
        // 28 * 7은 x축 / y축으로 2씩 증가되는 화소(1, 1 기준)
        // 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)의 제곱
        // 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)의 세제곱
        for (int x = 0; x < 14 * 14; ++x)
        {
          y += m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0] * m_W[c][k][x];
          y += m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1] * m_W[c][k][x + 14 * 14];
          y += pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 2) * m_W[c][k][x + 14 * 14 * 2];
          y += pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 3) * m_W[c][k][x + 14 * 14 * 3];
        }
      }
      else
      {
        // 분류기 3:
        // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)
        // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)의 주위 4개의 평균
        // 28*7은 x축/y축으로 2씩 증가되는 화소의 차이((0, 0 기준과 1,1 기준)
        // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)의 제곱
        for (int x = 0; x < 14 * 14; ++x)
          y += m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0] * m_W[c][k][x];

        for (int x = 0; x < 14 * 14; ++x)
        {
          double lt = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
          double lb = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 1];
          double rt = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 0];
          double rb = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
          double mean = (lt + lb + rt + rb) / 4;

          y += mean * m_W[c][k][x + 14 * 14];
        }

        for (int x = 0; x < 14 * 14; ++x)
        {
          double lt = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
          double rb = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
          double interval = abs(lt - rb);

          y += interval * m_W[c][k][x + 14 * 14 * 2];
        }

        for (int x = 0; x < 14 * 14; ++x)
          y += pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 2) * m_W[c][k][x + 14 * 14 * 3];
      }
```

```
  // 바이어스 추가
  y += m_B[c][k];
  // 시그모이드 함수로 변환
  y = 1. / (1 + exp(-y));

  // 실제 값 & 손실 값 계산
  double d = OutputList[i - n_Index][k];
  double d2 = (d - y) * (d - y) * 0.5;
  Loss += d2;

  // 예측값 저장
  y_perceptron[k] = y;

  // Convolutional Layer 1
  // 분류기 1: 28*28의 모든 화소를 입력을 받음
  if (c == 0)
  {
    for (int x = 0; x < 28 * 28; ++x)
      m_W[c][k][x] += lr * (d - y) * y * (1 - y) * m_MnistTrainInput[i][x];
  }
  // Convolutional Layer 2
  // 분류기 2:
  else if (c == 1)
  {
    for (int x = 0; x < 14 * 14; ++x)
    {

      // 1. 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)
      m_W[c][k][x] += lr * (d - y) * y * (1 - y) * m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
      // 2. 28 * 7은 x축 / y축으로 2씩 증가되는 화소(1, 1 기준)
      m_W[c][k][x + 14 * 14] += lr * (d - y) * y * (1 - y) * m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
      // 3. 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)의 제곱
      m_W[c][k][x + 14 * 14 * 2] += lr * (d - y) * y * (1 - y) * pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0],
      // 4. 28 * 7은 x축 / y축으로 2씩 증가되는 화소(0, 0 기준)의 세제곱
      m_W[c][k][x + 14 * 14 * 3] += lr * (d - y) * y * (1 - y) * pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0],
    }
  }
  // Convolutional Layer 3
  // 분류기 3:
  // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)
  // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)의 주위 4개의 평균
  // 28*7은 x축/y축으로 2씩 증가되는 화소의 차이((0, 0 기준과 1,1 기준)
  // 28*7은 x축/y축으로 2씩 증가되는 화소(0, 0 기준)의 제곱
  else
  {
    for (int x = 0; x < 14 * 14; ++x)
      m_W[c][k][x] += lr * (d - y) * y * (1 - y) * m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];

    for (int x = 0; x < 14 * 14; ++x)
    {
      double lt = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
      double lb = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 1];
      double rt = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 0];
      double rb = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
      double mean = (lt + lb + rt + rb) / 4;

      m_W[c][k][x + 14 * 14] += lr * (d - y) * y * (1 - y) * mean;
    }

    for (int x = 0; x < 14 * 14; ++x)
    {
      double lt = m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
      double rb = m_MnistTrainInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
      double interval = abs(lt - rb);

      m_W[c][k][x + 14 * 14 * 2] += lr * (d - y) * y * (1 - y) * interval;
    }

    for (int x = 0; x < 14 * 14; ++x)
      m_W[c][k][x + 14 * 14 * 3] += lr * (d - y) * y * (1 - y) * pow(m_MnistTrainInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0],
  }
  // 바이어스 업데이트
  m_B[c][k] += lr * (d - y) * y * (1 - y);
}

int MAX_value = -1;   // 예측값 중 가장 큰 값의 인덱스를 저장하기 위한 변수 초기화
double max_value = -100;   // 예측값 중 가장 큰 값 저장하기 위한 변수 초기화
for (int k = 0; k < 10; k++)
{
  if (y_perceptron[k] > max_value)   // 예측값이 현재까지의 최댓값보다 크다면
  {
    max_value = y_perceptron[k];   // 최댓값을 업데이트
    MAX_value = k;   // 최댓값의 인덱스를 저장
  }
}
int d = -2;   // 실제 정답값의 인덱스를 저장하기 위한 변수 초기화
```

```
    for (int k = 0; k < 10; k++)
    {
      if (OutputList[i - n_Index][k] == 1)   // 실제 정답값이면
        d = k;   // 정답값의 인덱스를 저장
    }

    if (MAX_value == d)   // 예측값과 실제 정답값의 인덱스가 일치하면
      nTP++;   // True Positive(TP) 개수를 증가시킴
  }
```

예측값과 실제 정답값이 일치하는 경우 True Positive(TP) 개수를 증가

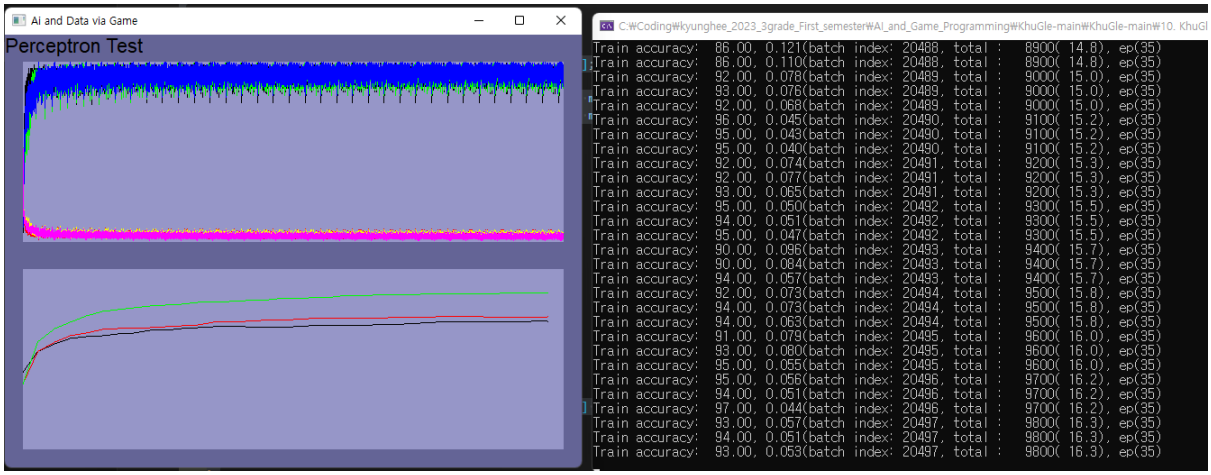
```
int MAX_value = -1;   // 예측값 중 가장 큰 값의 인덱스를 저장하기 위한 변수 초기화
double max_value = -100;   // 예측값 중 가장 큰 값 저장하기 위한 변수 초기화
for (int k = 0; k < 10; k++)
{
  if (y_perceptron[k] > max_value)   // 예측값이 현재까지의 최댓값보다 크다면
  {
    max_value = y_perceptron[k];   // 최댓값을 업데이트
    MAX_value = k;   // 최댓값의 인덱스를 저장
  }
}
int check = -2;   // 실제 정답값의 인덱스를 저장하기 위한 변수 초기화
for (int k = 0; k < 10; k++)
{
  if (OutputList[i - n_Index][k] == 1)   // 실제 정답값이면
    check = k;   // 정답값의 인덱스를 저장
}

if (MAX_value == check)   // 예측값과 실제 정답값의 인덱스가 일치하면
  nTP++;   // True Positive(TP) 개수를 증가시킴
```

테스터 데이터 값 정확도 계산 및 출력

```
if (n_Index + m_nBatch == m_nMnistTrainTotal)
{
  // 첫 번째 컬러 채널인 경우, epoch 카운트를 증가시킴
  if (c == 0) m_nEpochCnt++;

  // True Positive(TP) 개수를 저장하기 위한 변수 초기화
  int nTP = 0;
  int i;

  for (i = 0; i < m_nMnistTestTotal; i++)
  {
    double y_perceptron[10];
    for (int k = 0; k < 10; k++)
    {
      double y = 0;

      if (c == 0)
      {
        for (int x = 0; x < 28 * 28; ++x)
          y += m_MnistTestInput[i][x] * m_W[c][k][x];
      }
      else if (c == 1)
      {
        for (int x = 0; x < 14 * 14; ++x)
        {
          y += m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0] * m_W[c][k][x];
          y += m_MnistTestInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1] * m_W[c][k][x + 14 * 14];
          y += pow(m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 2) * m_W[c][k][x + 14 * 14 * 2];
          y += pow(m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 3) * m_W[c][k][x + 14 * 14 * 3];
        }
      }
      else
      {
        for (int x = 0; x < 14 * 14; ++x)
          y += m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0] * m_W[c][k][x];
```

```
        for (int x = 0; x < 14 * 14; ++x)
        {
          double lt = m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
          double lb = m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 1];
          double rt = m_MnistTestInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 0];
          double rb = m_MnistTestInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
          double mean = (lt + lb + rt + rb) / 4;

          y += mean * m_W[c][k][x + 14 * 14];
        }

        for (int x = 0; x < 14 * 14; ++x)
        {
          double lt = m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0];
          double rb = m_MnistTestInput[i][(x / 14 * 2 + 1) * 28 + x % 14 * 2 + 1];
          double interval = abs(lt - rb);

          y += interval * m_W[c][k][x + 14 * 14 * 2];
        }

        for (int x = 0; x < 14 * 14; ++x)
          y += pow(m_MnistTestInput[i][(x / 14 * 2 + 0) * 28 + x % 14 * 2 + 0], 2) * m_W[c][k][x + 14 * 14 * 3];
      }

      y += m_B[c][k];
      y = 1. / (1 + exp(-y));

      y_perceptron[k] = y;
    }

    // 예측값 중 가장 큰 값의 인덱스를 저장하기 위한 변수 초기화
    int MAX_value = -1;
    // 예측값 중 가장 큰 값 저장하기 위한 변수 초기화
    double max_value = -100;
    for (int k = 0; k < 10; k++)
    {
      if (y_perceptron[k] > max_value)
      {
        // 최댓값 저장
        max_value = y_perceptron[k];
        MAX_value = k;
      }
    }

    // 예측값의 인덱스를 결과값으로 설정
    int nResult = MAX_value;
    // 예측값과 실제 정답값이 일치하면 True Positive(TP) 개수를 증가시킴
    if (m_MnistTestOutput[i] == nResult) nTP++;
  }

  sprintf(Msg, "Test accuracy: %7.3lf\n", (double)nTP / (double)m_nMnistTestTotal * 100.);
  std::cout << Msg << std::endl;

  m_pTestGraphLayer->m_Data[c].push_back((((double)nTP / (double)m_nMnistTestTotal * 100 / 10) - 8.5) * 100);

  // 세 번째 컬러 채널인 경우
  if (c == 2)
  {
    // 그래프 레이어의 현재 개수를 증가시킴
    m_pTestGraphLayer->m_nCurrentCnt++;
    m_pTestGraphLayer->DrawBackgroundImage();
  }
}
}
```

결론



검정과 나머지 차이에 대해서

검정 분류기는 0은 입력 데이터의 구조와 패턴을 고려하지 않고 모든 화소 값을 동등하게 처리하므로 성능이 낮다. 이는 다른 값에 대해서 전혀 고려하지 않고 자기 자신만 고려하기 때문에 그렇다.

빨강과 초록의 차이에 대해서

초기에는 결과값이 잘 나오지 않지만 시간이 많이 지나면 녹색 적색 검정색 분류기의 순서대로 정렬이 되는 데 이는 3번 분류기의 성능이 뛰어나게 좋다. 이는 주변의 데이터의 평균과 차이에 대해서 고려하기 때문에 더 많은 데이터에 대해서 고려하게 되고 이는 성능이 좋다는 결론을 도출하게 된다.