Пример:

1.1. Преобразование изображений.

Начнём с преобразования mnistовских изображений.

В mnist изображения инициализированы, как (белый по чёрному).

Значит инициализируем каждый пиксель как (0 – чёрный, 1 - белый).

Подаются пиксели изображения в виде цифр от 0 до 255.

Этими строками преобразовываем пиксели изображения в тип double от 0 до 1.

```
for (int i=0; i < image_size; i++){
    binary_image[i]=(255.0 - (double) image_data[i]) / 255.0;
}
```

То есть, из 255 вычитаем значение пикселя по определённому индексу и разность делим на 255 для получения числа в диапазоне от 0 до 1.

Получаем на выходе 50 тыс изображений размером 28*28. Каждый пиксель каждого изображения имеет тип double и находится в диапазоне от 0 до 1.

1.2. Фильтрация изображений.

Фильтрация или свёртка применяется для выявления взаимосвязей между пикселями. Взаимосвязи могут выявляться между соседними пикселями по вертикали, горизонтали, диагонали и другим расположениям. Но в нашем случае, будет производиться выявление признаков между соседними пикселями по вертикали и горизонтали. Таким образом ускоряется обучение модели.

Будут использоваться 2 фильтра размером 3*3:

```
// Фильтр 3x3
int filter1[9]={-1, -1, -1,
    0, 0, 0,
    1, 1, 1};
int filter2[9]={-1, 0, 1,
    -1, 0, 1,
    -1, 0, 1};
```

Происходит формирование полностью черного изображения размером 30*30, так как при фильтрации размер изображения будет уменьшен на 2 пикселя по вертикали и горизонтали. После чего внутреннюю часть заполним пикселями поступившего изображения. У нас получится черная цифра на белом фоне, но обведённая черным цветом:

```
double padded[30*30]={0.0};
for (int y=0; y < 28; y++){
    for (int x=0; x < 28; x++){
        padded[(y+1) * 30 + (x+1)]=image_data[y * 28 + x];
    }
}
```

Далее создаём 3 массива:

```
double filtered1[28 * 28]={0.0};
double filtered2[28 * 28]={0.0};
double combined[28 * 28]={0.0};
```

Для чего они нужны, объясню позже:

Дальше происходит свёртка:

```
// Применение свёртки
for (int y=0; y < 28; y++){
    for (int x=0; x < 28; x++){
        double sum1=0.0;
        double sum2=0.0;
        // Проход по фильтру 3x3
        for (int ky=0; ky < 3; ky++){
            for (int kx=0; kx < 3; kx++){
                sum1+=padded[(y+ky) * 30 + (x+kx)] * filter1[ky * 3 + kx];
                sum2+=padded[(y+ky) * 30 + (x+kx)] * filter2[ky * 3 + kx];
            }
        }
        filtered1[y * 28 + x]=sum1;
        filtered2[y * 28 + x]=sum2;
    }
}
```

Здесь происходит пропускание двух матриц filter1 и filter2 по изображению и подсчёт сумм элементов. Каждый элемент представляет собой произведение компонента матрицы и пикселя изображения, напротив которого лежит этот компонент матрицы. То есть матрица накладывается на начало изображения, выполняет вышеописанную операцию и двигается дальше до конца изображения.

В данном случае шаг матрицы == 1.

Рассмотрим, как происходит фильтрация изображения после этого места:

```
double padded[30*30]={0.0};
for (int y=0; y < 28; y++){
    for (int x=0; x < 28; x++){
        padded[(y+1) * 30 + (x+1)]=image_data[y * 28 + x];
    }
}
```

```
// Применение свёртки
for (int y=0; y < 28; y++){
    for (int x=0; x < 28; x++){
        double sum1=0.0;
        double sum2=0.0;
        // Проход по фильтру 3x3
        for (int ky=0; ky < 3; ky++){
            for (int kx=0; kx < 3; kx++){
                sum1+=padded[(y+ky) * 30 + (x+kx)] * filter1[ky * 3 + kx];
                sum2+=padded[(y+ky) * 30 + (x+kx)] * filter2[ky * 3 + kx];
            }
        }
        filtered1[y * 28 + x]=sum1;
        filtered2[y * 28 + x]=sum2;
    }
}
```

Предположим, у нас такое изображение, где границы заполнены нулями, фон – единицами, а цифра нулями:
(На самом деле у нас изображения не строго бинаризованные. Просто здесь я привожу пример для понимания.)

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 0 1 1 0 1 1 1 1 1 0
0 1 1 1 1 0 1 1 0 1 1 1 1 1 0
0 1 1 1 1 0 0 0 0 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 1 1 1 1 1 0
0 1 1 1 1 1 1 1 0 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1. Возьмём часть самой верхней границы изображения 30*30:

0 0 0
1 1 1
1 1 1
Применим горизонтальный фильтр Собеля:
-1 -1 -1
0 0 0
1 1 1
получается 1*1 + 1*1 + 1*1 == 3. Верхняя граница изображения.
То есть первая строка массива filtered1 будет содержать {3}.
Спускаемся по фону:
1 1 1
1 1 1
1 1 1
При наложении фильтра получаем:
-1*1 + -1*1 + -1*1
+ 0*1 + 0*1 + 0*1
+ 1*1 + 1*1 + 1*1
== 0 – это фон в массиве filtered1, а так же и в filtered2 получится черный, а граница == 3 – это лишний артефакт, который во время применения формулы:

```
    else{ combined[i]=sqrt(filtered1[i] * filtered1[i] + filtered2[i] * filtered2[i]); }
```

будет вызывать проблему на границе изображения. Это надо будет устранить после фильтрации вот таким образом:

```
if (i<28 || i>755 || i%28==0 || i%28==27){ combined[i]=0.0; }
```

А вот вся эта часть:

```
for (int i=0; i < 28*28; i++){
    if (i<28 || i>755 || i%28==0 || i%28==27){ combined[i]=0.0; }
    else{ combined[i]=sqrt(filtered1[i] * filtered1[i] + filtered2[i] * filtered2[i]); }
}
```

Возьмём ещё раз часть верхней границы изображения, чтобы проверить, как уже вертикальный фильтр будет с ней работать:
0 0 0
1 1 1
1 1 1

Применим вертикальный фильтр Собеля:
-1 0 1
-1 0 1
-1 0 1
При наложении здесь получается -1*1 + 0*1 + 1*1 == 0. Нижняя граница изображения. Тут ничего не нарушается, поэтому за этот момент можно не переживать.

Примечание: тот же самый эффект будет происходить с горизонтальной границей изображения при проходе по ней вертикального фильтра.

А дальше происходит та самая очистка границ – приведение их к черному цвету.

```
if (i<28 || i>755 || i%28==0 || i%28==27){ combined[i]=0.0; }
```

А так же, для остальной части изображения происходит вычисление квадратного корня от суммы квадратов элементов массивов filtered1 и filtered2.

```
for (int i=0; i < 28*28; i++){
    if (i<28 || i>755 || i%28==0 || i%28==27){ combined[i]=0.0; }
    else{ combined[i]=sqrt(filtered1[i] * filtered1[i] + filtered2[i] * filtered2[i]); }
}
```

Это вычисление градиента яркости (величины изменения) в каждой точке изображения.
Почему именно такая формула?
filtered1[i] и filtered2[i] содержат значения изменения яркости по вертикали и горизонтали.
Формула combined[i] = sqrt(filtered1[i]² + filtered2[i]²) — это длина вектора градиента в точке изображения.

- filtered1[i] (Gx)— изменение яркости по горизонтали (ось X).
  - filtered1[i] > 0 → яркость растёт справа налево.
  - filtered1[i] < 0 → яркость растёт слева направо.
- filtered2[i] (Gy) — изменение яркости по вертикали (ось Y).
  - filtered2[i] > 0 → яркость растёт снизу вверх.
  - filtered2[i] < 0 → яркость растёт сверху вниз.

sqrt(Gx² + Gy²) – это формула теоремы Пифагора.
filtered1 и filtered2 содержат данные об одной и той же части, но это информация о горизонтальном изменении яркости и вертикальном.
Это способ комбинирования результатов двух ортогональных фильтров Собеля.
Длина вектора в двумерном пространстве вычисляется по теореме Пифагора.

Формула sqrt(Gx² + Gy²) — это математический способ вычисления длины вектора градиента в обработке изображений.

```
for (int i=0; i < 28*28; i++){
    if (i<28 || i>755 || i%28==0 || i%28==27){ combined[i]=0.0; }
    else{ combined[i]=sqrt(filtered1[i] * filtered1[i] + filtered2[i] * filtered2[i]); }
}
```

После этого, мы делаем инверсию – возвращаем исходные цвета. То есть делаем изображения в цифровом формате в виде "черным по белому":

```
for (int i=0; i < 28*28; i++){
    combined[i]=1.0 - combined[i];  // Инверсия: фон станет белым, а цифра снова черной
}
```

1.3. Пуллинг изображений.

Нужно произвести пуллинг.

Вопрос: для чего? Почему сразу не добавил свёрточные слои? Почему не обучаю модель на оригинальных изображениях 28*28?

Ответ: это тестовая практика для тестирования быстрого результата. Тестов проводить нужно много. Если каждый раз обучать на оригинальных изображениях, уйдёт слишком много времени. Если сразу добавить свёрточные слои, можно запутаться в слоях. После обучения без фильтрации уже добавлен свёрточный слой и модель была переформирована.

Но теперь уже свёрточный слой добавлен и описан выше, поэтому перейдём к обучению.

Изначально было 10 обучающих итераций. Шаг обучения = 0.01. Это слишком быстро, поэтому модель не обучалась, а веса взрывались. Становились равными огромным числам.

Я увеличил количество обучающих итераций до 100 и понизил шаг обучения до 0.000001. Модель стала обучаться правильно, но дольше.

Проведено большое количество тестов. Чтобы сэкономить время, я отпуллинговал изображения и на основании отпуллингованных изображений обучил модель.

Начинаем:

Мы имеем бинарный файл, в котором содержится 50тыс изображений размером 28*28. Каждый пиксель каждого изображения имеет тип double и находится в диапазоне от 0 до 1.

Берём одно изображение и пропускаем по нему матрицу размером 2*2. Ищем самый чёрный пиксель по каждому участку размером 2*2 и сохраняем его в отдельный массив, который потом выносим в бинарный файл.

```
for (int y=0; y<28; y+=2){
    for (int x=0; x<28; x+=2){
        max[s]=2.0;
        for (int ky=0; ky<2; ky++){
            for (int kx=0; kx<2; kx++){
                if (image[(y+ky) * 28 + (x+kx)]<max[s]){ max[s]=image[(y+ky) * 28 + (x+kx)]; }
            }
        }
        s++;
    }
}
```

То же самое делаем со всеми остальными изображениями.

В итоге получаем 50 тыс изображений размером 14*14.

1.3. Формирование компонентов, которые мы будем обучать.

1. Мы имеем бинарный файл, в котором содержится 50тыс отпуллингованных изображений цифр от 0 до 9 размером 14*14.
2. Обозначим каждое изображение размером 14*14, взятое из бинарного файла, как вектор (x), инициализированный размером 14*14=[196].
3. Формируем случайные значения следующих компонентов:
    a. Один единый набор весов (W), который мы будем обучать. Размер [196][10].

```
for (int m=0; m < 10; m++){
    for (int n=0; n < 196; n++){
        W[n][m]=((double) rand() / RAND_MAX) * 0.2 - 0.1;
    }
}
```

    b. Один единый вектор коэффициентов смещения (b), который мы будем обучать. Размер [10].

```
for (int n=0; n < 10; n++){ b[n]=0.01; }
```

    c. Сырые логиты – вектор (z) размером [10], вычисленный на основании случайных весов, матрицы (x) и коэффициентов смещения. То есть, для каждого изображения свой вектор. Вектора получаются не одинаковые, так как мы формируем каждый вектор на основании разных векторов (x) размером [196].

```
// Вычисление сырых логитов
double z[10]={0};
for (int m=0; m<10; m++){
    for (int n=0; n<196; n++){
        z[m]+=x[n] * W[n][m];  // Матричное умножение
    }
    z[m]+=b[m];  // Добавляем смещение
}
```

4. Ищем максимальное значение среди 10 сырых логитов.

```
// Нахождение максимального значения
// для числовой стабильности softmax
double max=z[0];
for (int m=1; m<10; m++){
    if (z[m] > max){ max=z[m]; }
}
```

5. Применяем нормализацию для избежания возведения экспоненты в огромные числа. Экспоненцирование нужно только для того, чтобы привести все элементы будущего вектора (y) к положительным значениям в диапазон от 0 до 1. Так как значения могут быть как положительными, так и отрицательными.

После чего, формируем сумму, состоящую из экспонент в степени разности сырого логита и максимального сырого логита.

```
// Экспоненцирование разности сырых логитов
// и максимального значения -
// СТАБИЛИЗАЦИЯ и подготовка к softmax
double sum=0.0;
for (int m=0; m<10; m++){
    yy[m]=exp(z[m] - max);
    sum+=yy[m];
}
```

В нашем случае с распознаванием цифр нормализация (z[m] - max) не нужна, так как мы работаем с пикселями, каждый из которых находится в диапазоне от 0 до 1. Весовые коэффициенты и коэффициенты смещения тоже не будут больше (1) или меньше (-1). Но по регламенту не будем удалять нормализацию. Так как в дальнейшем при работе с более сложными изображениями сырые логиты (z) могут оказаться довольно высокими из-за другого формата пикселей (x) (не чёрно-белый формат).
Например зелёный-красный-синий, где оттенок каждого цвета имеет цифру от 0 до 255.

Казалось бы, вектор (x) не имеет отрицательных значений, коэффициент b тоже. А вот весовые коэффициенты W могут иметь отрицательные числа. А так же, по формуле с использованием градиентного спуска могут вычитаться отрицательные числа в случае попадания на элемент вектора (y), взятый по индексу, равному взятой метке, а могут вычитаться положительные числа в противном случае.
Именно по этой причине сырые логиты (z) могут получаться отрицательные. А значит мы, должны применить экспоненцирование.

```
for (int m=0; m < 10; m++){
    double grad=y[m] - (m == label?1.0:0.0);
    for (int n=0; n < 196; n++){
        W[n][m]-=learning_rate * (grad * x[n]);
    }
    b[m]-=learning_rate * grad;
}
```

Но к этой части мы вернёмся позже.

6. Создаём вектор (y), который содержит в себе 10 случайных вероятностных коэффициентов.

```
// А это уже самый настоящий softmax (нормализация)
for (int m=0; m<10; m++){ yy[m]/=sum; }
```

1.4. Обучение.

Ничего ещё не обучено.

1. Чтобы обучить модель, нужно отрегулировать веса (W) и коэффициент смещений (b).

```
for (int m=0; m < 10; m++){
    double grad=y[m] - (m == label?1.0:0.0);
    for (int n=0; n < 196; n++){
        W[n][m]-=learning_rate * (grad * x[n]);
    }
    b[m]-=learning_rate * grad;
}
```

2. Чтобы отрегулировать веса (W) и коэффициент смещений (b), нужно вычислить градиент.

```
double grad=y[m] - (m == label?1.0:0.0);
```

В машинном обучении, градиент функции потерь – это вектор, который указывает направление наиболее быстрого возрастания этой функции в заданной точке (в пространстве параметров модели). Другими словами, если мы хотим уменьшить функцию потерь (что является нашей целью при обучении), нам нужно двигаться в направлении, противоположном градиенту. Этот процесс называется градиентным спуском.

3. А чтобы вычислить градиент, нужно провести математические операции с функцией потерь.

```
loss=-log(y[label] + 1e-10);
```

Поэтому, первым делом мы вычисляем логарифмическую функцию потерь.

В данном случае выделен элемент вектора (y) под индексом (0).

Y={0.210644, 0.044656, 0.067404, 0.083717, 0.078198, 0.108800, 0.123682, 0.084028, 0.082621, 0.116250}

Мы берём из бинарного файла вектор (x) размером [196] и принадлежащую ему метку label.
Если на изображении изображён (0), то и label==(0).

Вычисляем логарифмическую функцию потерь по тому индексу элемента вектора (y), метку которого мы взяли из бинарного файла. То есть label = i. Нам не нужны остальные элементы, потому что разработчик сам лично указывает программе, по какому индексу брать элемент вектора (y).

Для чего нужна функция потерь?

Казалось бы, в самой формуле градиентного спуска функция потерь напрямую никак не участвует.
Но по функции потерь берётся комбинированная производная и уже результат производной используется в формуле градиентного спуска.

В данном случае, нам она наглядно показывает, обучается ли модель. Если функция потерь падает, значит модель обучается. Чем ближе значение вероятности к единице, но меньше единицы, тем меньше получается функция потерь.

Для обучения нашей модели мы используем логарифмическую функцию потерь.
Вопрос:
Почему именно логарифмическая и почему впереди стоит знак минус (-), для чего (1e-1)?

```
loss=-log(y[label] + 1e-10);
```

Ответ:
Работать нам приходится со значениями меньше единицы (поэтому берётся логарифмическая функция потерь).
Судя по математическим свойствам логарифма, чем меньше аргумент, от которого берётся логарифм, тем выше возрастает значение, логарифма, а именно, функции потерь, но в отрицательном направлении (поэтому, чтобы функция потерь была положительной, вручную перед логарифмом ставим минус).
То есть, мы берём логарифм по элементу вектора (y) и этот элемент стоит под индексом (label).
Если этот элемент вектора (y) близок к нулю, функция потерь будет большая по модулю.
Если же, этот элемент вектора (y) близок к единице, то функция потерь будет низкая, а значит, модель вышла на путь правильного обучения.

(1e-1) – нужна для численной стабильности, если появится слишком маленькое дробное число, которое компилятор примет за 0.

1.5. А дальше предстоит большая работа с функцией потерь для вычисления градиента.

Что бы узнать направление и крутизну логарифмической функции потерь, мы должны взять производную.

Вопрос: производную от чего?

Единственный изменяемый элемент функции потерь – вектор (y).

Вектор (y) состоит из формулы

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Вопрос: почему мы возводим экспоненту в степень сырого логита?
Ответ: потому что сырой логит может получиться любым числом: отрицательное, положительное. Поэтому мы должны привести все числа к положительному виду для вычисления вероятности.

Тут единственный изменяемый элемент – это сырой логит (z).

z = Wx + b.

А вот от (Z) мы уже можем взять производную по (W).

Нам предстоит взять комбинированную производную по 3-м компонентам.

2.1. Берём производную от основной функции потерь по вектору (y)

$$L = -\sum_{k=1}^{K} t_k \log y_k$$

$$\frac{\partial L}{\partial y_j} = -\sum_{k=1}^{K} t_k \cdot \begin{cases} \frac{1}{y_j} & \text{при } k = j \\ 0 & \text{при } k \neq j \end{cases}$$

2.2. Берём производную от softmax - вектора (y) по сырым логитам (z)

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} = \frac{e^{z_j}}{S}, \quad \text{где } S = \sum_{k=1}^{K} e^{z_k}$$

$$\begin{aligned}
\frac{\partial y_j}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{e^{z_j}}{S}\right) \\
&= \frac{e^{z_j} S - e^{z_j} \frac{\partial S}{\partial z_j}}{S^2} \\
&= \frac{e^{z_j} S - e^{z_j} e^{z_j}}{S^2} \qquad \left(\text{так как } \frac{\partial S}{\partial z_j} = e^{z_j}\right) \\
&= \frac{e^{z_j}(S - e^{z_j})}{S^2} \\
&= \frac{e^{z_j}}{S} \cdot \frac{S - e^{z_j}}{S} \\
&= y_j(1 - y_j)
\end{aligned}$$

Примечание:

$$\left(\text{так как } \frac{\partial S}{\partial z_j} = e^{z_j}\right)$$

В функции потерь берётся логарифм по элементу вектора (y) только под тем индексом, который равен взятой метке.

Поэтому происходит взятие частной производной. А экпоненты в степенях под другими индексами при взятии производной по логиту под индексом, равным взятой метке, автоматически обнуляются. Остаётся только та экспонента, индекс степени которой равен взятой метке.

2.3. Берём производную от сырого логита (z) по обучаемым весам (W)

$$z_j = \sum_{i=1}^{n} W_{ij} x_i + b_j$$

$$\frac{\partial z_j}{\partial W_{ij}} = x_i$$

Но в то же время:

$$\frac{\partial z_j}{\partial b_j} = 1$$

3. Комбинируем найденные производные.

$$\frac{\partial L}{\partial y_j} = -\frac{1}{y_j}$$

$$\frac{\partial y_j}{\partial z_j} = y_j(t_j - y_j)$$

$$\frac{\partial z_j}{\partial W_{ij}} = x_i$$

Итог:

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial W_{ij}} = -\frac{1}{y_j} \cdot y_j(t_j - y_j) \cdot x_i = (y_j - 1) \cdot x_i$$

где $t_j = 1$ при $j = label$

В то же время производная от функции потерь берётся и по коэффициенту смещения:

$$\frac{\partial L}{\partial b_j} = \frac{\partial L}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j} = -\frac{1}{y_j} \cdot y_j(t_j - y_j) \cdot x_i = (y_j - 1) \cdot 1$$

А теперь мы можем вычислить градиент:

```
double grad=y[m] - (m == label?1.0:0.0);
```

(y[m]-1) – это и есть градиент.

3.1. Обучение весов и коэффициентов смещения.

```
double learning_rate=0.000001;
for (int m=0; m < 10; m++){
    double grad=y[m] - (m == label?1.0:0.0);
    for (int n=0; n < 196; n++){
        W[n][m]-=learning_rate * (grad * x[n]);
    }
    b[m]-=learning_rate * grad;
}
```

Помимо всего прочего, мы видим переменную learning_rate. Это шаг обучения. Чем он ниже, чем тщательнее обучение, но оно становится медленнее.

В идеале, грамотное обучение формируется при помощи короткого шага обучения и большого количества обучающих итераций.

На каждой обучающей итерации обновляются веса и коэффициенты смещения. Но вектора (y) формируются заново за счёт нашего линейного уравнения. И на основе новых векторов (y) формируется новая функция потерь, за счёт чего происходит регулировка весов (W) и коэффициентов смещения (b).

4. Подача тестовых изображений на вход системе:

И наконец, на вход системе тестовое изображение должно подаваться ровно в таком виде, в котором подавались обучающие изображения в функцию обучения. То есть пропущенные через фильтры и пуллинг.