

Let's Build a CNN from Scratch Using C# 12

A convolutional neural network (CNN) is a long long road. The first step is to understand the neural network (NN).

```
// 1. define neural network
int[] net = { 784, 100, 100, 10 };
```

The first layer is the input layer and consists of 784 neurons. The next 2 layers with 100 neurons are the hidden layers. Finally, the last layer is the output layer with 10 neurons where the prediction comes out. It would be called an NN with 3 connected layers (in-h1, h1-h2, h2-out) that must be calculated.

I have to apologize for using “neurons” as a term for: units-nodes-perceptrons-cells. It is similar to the misleading term "neural network", or the systems we call AI (ChatGPT) since the end of 2022. Dealing with terminology alerts is a part of the game. What is similar is that all these types of systems are deep learning systems.

Where this network would be named logistic regression, a machine learning model:

```
int[] net = { 784, 10 };
```

A system with 2 hidden layers is already called a deep neural network, a deep learning system.

Let's start with the basic steps on a high level to build an NN and the first neat trick:

```
// 1. define neural network
// 2. initialize random weights
// 3. forward pass (FF)
// 4. backward pass (BP)
// 5. update weights
```

The Trick is Template Code

```
// template code
/*
// i = connected layer, j = input neurons, k = output neurons, m = layer weights
for (int i = 0, j = 0, k = net[0], m = 0; i < net.Length - 1; i++)
{
    // left input neurons length, right output neurons length
    int left = net[i], right = net[i + 1];
    for (int l = 0, w = m; l < left; l++)
    {
        for (int r = 0; r < right; r++)
        {
            // do stuff
        }
        w += right; // for bp needed, ff can be simplified
    }
    // stack weights, input and output id steps
    m += left * right; j += left; k += right;
}
*/
```

The code template can be used for all functions we need to run the NN with small changes. A deep understanding of this code example is almost the entire work and minimizes problems.

The Math

```
// feed forward (get prediction):
neuronOutputRight += neuronInputLeft * weight
// output error distance (target - output):
gradientOutputRight = target - neuronOutputRight
// backprop input gradient sum (go distance):
gradientInputLeft += weight * gradientOutputRight
// backprop weightGradient (go distance):
weightGradient += inputNeuronLeft * gradientOutputRight
// update (use small part e.g. 0.001):
weight += weightGradient * learningRate
// momentum (keep part of distance e.g. 0.5):
weightGradient *= momentum
```

This is all we need to know from the implementation perspective. We can now copy the template code 5 times and build the logic around to run the network.

From Forward Pass

```
// template forward layerwise
/*
int i = 0, j = 0, k = net[0], m = 0; // 1. from start

for (int i = 0; i < net.Length - 1; i++) // 2. forward
{
    int left = net[i], right = net[i + 1];

    for (int l = 0, w = m; l < left; l++, w += right)
        for (int r = 0; r < right; r++)
        {
            // do stuff
        }

    m += left * right; j += left; k += right; // 3. layer step up
}
*/
```

To Backward Pass

```
// template backward layerwise with 3 changes
/*
int j = neurons.Length - net[^1], k = neurons.Length, m = weights.Length; // 1. from end

for (int i = net.Length - 2; i >= 0; i--) // 2. backward
{
    int left = net[i], right = net[i + 1];
    m -= right * left; j -= left; k -= right; // 3. layer step down

    for (int l = 0, w = m; l < left; l++, w += right)
        for (int r = 0; r < right; r++)
        {
            // do stuff
        }
}
*/
```

```

// Glorot random weights init (template 0 and 1)
static float[] WeightsInitGlorot(int[] net, int seed)
{
    // get weights length (template 0)
    int len = 0;
    for (int i = 0; i < net.Length - 1; i++) // each layer
        for (int l = 0, left = net[i], right = net[i + 1] ; l < left; l++) // input neurons
            for (int r = 0; r < right; r++) len++; // output neurons
    // allocate memory
    float[] weights = new float[len];
    // set user seed
    Random rnd = new(seed);
    // Glorot random weights init (template 1)
    for (int i = 0, w = 0; i < net.Length - 1; i++) // each layer
    {
        int left = net[i], right = net[i + 1];
        float sd = MathF.Sqrt(6.0f / (net[i] + net[i + 1]));
        for (int l = 0; l < left; l++, w += right) // input neurons
        {
            for (int r = 0; r < right; r++) // output neurons
                weights[w + r] = rnd.NextSingle() * sd * 2 - sd;
        }
    }
    return weights;
}

// forward propagation (template 2)
static void FeedForward(float[] neurons, int[] net, float[] weights)
{
    for (int i = 0, j = 0, k = net[0], m = 0; i < net.Length - 1; i++) // each layer
    {
        int left = net[i], right = net[i + 1];
        for (int l = 0, w = m; l < left; l++, w += right) // input neurons
        {
            float n = neurons[j + 1];
            if (n > 0) // ReLU pre-activation
                for (int r = 0; r < right; r++) // output neurons
                    neurons[k + r] += n * weights[w + r];
        }
        m += left * right; j += left; k += right;
    }
}

```

```

// backpropagation (template 3)
static void Backprop(float[] neurons, int[] net, float[] weights, float[] weightGradients)
{
    for (int i = net.Length - 2, j = neurons.Length - net[i],
        k = neurons.Length, m = weights.Length; i >= 0; i--) // layers
    {
        int left = net[i], right = net[i + 1];
        m -= right * left; j -= left; k -= right;
        for (int l = 0, w = m; l < left; l++, w += right) // input neurons
        {
            float inputGradient = 0, n = neurons[j + 1];
            if (n > 0) // ReLU derivative
                for (int r = 0; r < right; r++) // output neurons
                {
                    var gradient = neurons[k + r];
                    inputGradient += weights[w + r] * gradient;
                    weightGradients[w + r] += n * gradient;
                }
            neurons[j + 1] = inputGradient;
        }
    }
}

// Stochastic Gradient Descent or Mini-batch-GD or Batch-GD (template 4)
static void Update(int[] net, float[] weights, float[] delta, float lr, float mom)
{
    for (int i = 0, w = 0; i < net.Length - 1; i++) // layers
    {
        int left = net[i], right = net[i + 1];
        for (int l = 0; l < left; l++, w += right) // input neurons
        {
            for (int r = 0; r < right; r++) // output neurons
            {
                weights[w + r] += delta[w + r] * lr;
                delta[w + r] *= mom;
            }
        }
    }
}

```

```

// helper functions
static float[] FeedSample(float[] samplesTrainingF, int x, int neuronLen)
{
    // neurons (input+hidden+output)
    float[] neurons = new float[neuronLen];
    // dataset id
    int sampleID = x * 784;
    // copy sample to input layer
    for (int i = 0; i < 784; i++)
        neurons[i] = samplesTrainingF[sampleID + i];
    return neurons;
}

// argmax and softmax
static int SoftArgMax(Span<float> neurons)
{
    // argmax prediction
    int id = 0;
    float max = neurons[0];
    for (int i = 1; i < neurons.Length; i++)
        if (neurons[i] > max)
        {
            max = neurons[i];
            id = i;
        }
    // softmax activation
    float scale = 0;
    for (int n = 0; n < neurons.Length; n++) // max trick
        scale += neurons[n] = MathF.Exp((neurons[n] - max));
    for (int n = 0; n < neurons.Length; n++)
        neurons[n] /= scale; // pseudo probabilities now
    return id; // return nn prediction
}

// target - output: distance between what we want minus what we get
static void ErrorGradient(Span<float> neurons, int target)
{
    for (int i = 0; i < neurons.Length; i++)
        neurons[i] = target == i ? 1 - neurons[i] : -neurons[i];
}

```

```

static void RunTraining(int[] net, float[] weights, float[] data, byte[] labels, int
BATCHSIZE, int EPOCHS, float LEARNINGRATE, float MOMENTUM)
{
    Console.WriteLine($"Training progress:");
    float[] weightGradients = new float[weights.Length];
    int neuronLen = 0;
    for (int i = 0; i < net.Length; i++)neuronLen += net[i];
    for (int epoch = 0, B = labels.Length / BATCHSIZE; epoch < EPOCHS; epoch++) // each epoch
    {
        int correct = 0;
        for (int b = 0; b < B; b++) // each batch
        {
            for (int x = b * BATCHSIZE, X = (b + 1) * BATCHSIZE; x < X; x++) // each sample
            {
                float[] neurons = FeedSample(data, x, neuronLen);
                FeedForward(neurons, net, weights);
                var outs = neurons.AsSpan().Slice(neuronLen - net[^1], net[^1]);
                int prediction = SoftArgMax(outs); // reference output neurons
                int target = labels[x];
                correct += target == prediction ? 1 : 0;
                ErrorGradient(outs, target); // reference output neurons
                Backprop(neurons, net, weights, weightGradients);
            }
            Update(net, weights, weightGradients, LEARNINGRATE, MOMENTUM);
        }
        Console.WriteLine($"Epoch = {1 + epoch,2} | accuracy = {
            correct * 100.0 / (B * BATCHSIZE),5:F2}%");
    }
}
//
static void RunTesting(int[] net, float[] weights, float[] data, byte[]labels)
{
    int neuronLen = 0;
    for (int i = 0; i < net.Length; i++) neuronLen += net[i];
    int correct = 0;
    for (int x = 0; x < labels.Length; x++) { // each sample
        float[] neurons = FeedSample(data, x, neuronLen);
        FeedForward(neurons, net, weights);
        var outs = neurons.AsSpan().Slice(neuronLen - net[^1], net[^1]);
        int prediction = SoftArgMax(outs);
        int target = labels[x];
        correct += target == prediction ? 1 : 0;
    }
    Console.WriteLine($"Test accuracy = {(correct * 100.0 / labels.Length),6}%");
}

```

```
// run the code:
Console.WriteLine($"Begin basic neural network demo\n");
// get dataset
AutoData d = new(@"C:\basic_nn\", AutoData.Dataset.MNIST);
// define neural network
int[] net = { 784, 100, 100, 10 };
// get random weights
float[] weights = WeightsInitGlorot(net, 12345);
// nn training
RunTraining(net, weights, d.samplesTrainingF, d.labelsTraining, 100, 10, 0.001f, 0.5f);
// nn test
RunTesting(net, weights, d.samplesTestF, d.labelsTest);
```



```

struct AutoData {
    public byte[] labelsTraining, labelsTest;
    public float[] samplesTrainingF, samplesTestF;
    static float[] NormalizeData(byte[] samples) => samples.Select(s => s / 255f).ToArray();
    public AutoData(string path, Dataset datasetType) {
        byte[] test, training; // Define URLs and file paths based on dataset
        string trainDataUrl, trainLabelUrl, testDataUrl, testLabelUrl;
        string trainDataPath, trainLabelPath, testDataPath, testLabelPath;
        if (datasetType == Dataset.MNIST) {
            var baseMnistUrl = // Hardcoded URLs for MNIST data
                "https://github.com/grensens/gif_test/raw/master/MNIST_Data/";
            (trainDataUrl, trainLabelUrl, testDataUrl, testLabelUrl) = (
                $"{baseMnistUrl}train-images.idx3-ubyte",
                $"{baseMnistUrl}train-labels.idx1-ubyte",
                $"{baseMnistUrl}t10k-images.idx3-ubyte",
                $"{baseMnistUrl}t10k-labels.idx1-ubyte");
            (trainDataPath, trainLabelPath, testDataPath, testLabelPath) = (
                "trainData_MNIST", "trainLabel_MNIST",
                "testData_MNIST", "testLabel_MNIST");
        } else { // if (datasetType == DatasetType.FashionMNIST)
            var baseFashionUrl = // Hardcoded URLs for Fashion MNIST
                "https://github.com/zalandoresearch/fashion-mnist/raw/master/data/fashion/";
            (trainDataUrl, trainLabelUrl, testDataUrl, testLabelUrl) = (
                $"{baseFashionUrl}train-images-idx3-ubyte.gz",
                $"{baseFashionUrl}train-labels-idx1-ubyte.gz",
                $"{baseFashionUrl}t10k-images-idx3-ubyte.gz",
                $"{baseFashionUrl}t10k-labels-idx1-ubyte.gz"); // Paths for Fashion MNIST data
            (trainDataPath, trainLabelPath, testDataPath, testLabelPath) = (
                "trainData_FashionMNIST", "trainLabel_FashionMNIST",
                "testData_FashionMNIST", "testLabel_FashionMNIST");
        }
        if (!File.Exists(Path.Combine(path, trainDataPath))
            || !File.Exists(Path.Combine(path, trainLabelPath))
            || !File.Exists(Path.Combine(path, testDataPath))
            || !File.Exists(Path.Combine(path, testLabelPath))) {
            Console.WriteLine($"Status: {datasetType} data not found");
            if (!Directory.Exists(path)) Directory.CreateDirectory(path);
            // padding bits: data = 16, labels = 8
            Console.WriteLine("Action: Downloading data from GitHub");
            training = DownloadAndExtract(trainDataUrl, datasetType).Skip(16).Take(60000 * 784).ToArray();
            labelsTraining = DownloadAndExtract(trainLabelUrl, datasetType).Skip(8).Take(60000).ToArray();
            test = DownloadAndExtract(testDataUrl, datasetType).Skip(16).Take(10000 * 784).ToArray();
            labelsTest = DownloadAndExtract(testLabelUrl, datasetType).Skip(8).Take(10000).ToArray();
            Console.WriteLine("Save path: " + path + "\n");
            File.WriteAllBytesAsync(Path.Combine(path, trainDataPath), training);
            File.WriteAllBytesAsync(Path.Combine(path, trainLabelPath), labelsTraining);
            File.WriteAllBytesAsync(Path.Combine(path, testDataPath), test);
            File.WriteAllBytesAsync(Path.Combine(path, testLabelPath), labelsTest);
        } else {
            Console.WriteLine($"Dataset: {datasetType} ({path})" + "\n");
            training = File.ReadAllBytes(Path.Combine(path, trainDataPath)).Take(60000 * 784).ToArray();
            labelsTraining = File.ReadAllBytes(Path.Combine(path, trainLabelPath)).Take(60000).ToArray();
            test = File.ReadAllBytes(Path.Combine(path, testDataPath)).Take(10000 * 784).ToArray();
            labelsTest = File.ReadAllBytes(Path.Combine(path, testLabelPath)).Take(10000).ToArray();
        }
        samplesTrainingF = NormalizeData(training); samplesTestF = NormalizeData(test);
    }
    static byte[] DownloadAndExtract(string url, Dataset datasetType) {
        using var client = new HttpClient();
        using var responseStream = client.GetStreamAsync(url).Result;
        using var ms = new MemoryStream();
        if (datasetType == Dataset.FashionMNIST) using (var gzipStream = new GZipStream(responseStream, CompressionMode.Decompress))
            gzipStream.CopyTo(ms);
        else responseStream.CopyTo(ms);
        return ms.ToArray();
    }
    public enum Dataset { MNIST, FashionMNIST }
}

```

You can simply copy the code into .NET 8 console application in Visual Studio 2022, remove the whitespace, and run it.

```
Begin basic neural network demo

Dataset: MNIST (C:\relu_net\)

Training progress:
Epoch = 1 | accuracy = 90.59%
Epoch = 2 | accuracy = 95.83%
Epoch = 3 | accuracy = 96.96%
Epoch = 4 | accuracy = 97.59%
Epoch = 5 | accuracy = 98.06%
Epoch = 6 | accuracy = 98.42%
Epoch = 7 | accuracy = 98.73%
Epoch = 8 | accuracy = 98.96%
Epoch = 9 | accuracy = 99.16%
Epoch = 10 | accuracy = 99.38%

Test accuracy = 97.37%
```

So that was a good exercise and the preparation to understand how to build a convolutional neural network. We use the same tricks, make it even easier, and drop a lot more things. The CNN code can now vary, also we need to use much better ways to compute the CNN. Here are the most important parts to understand my workflow in C# for fast computing.

C# 12: Parallel Loop, Spans and SIMD

Computing power is key to running a proper CNN. Instead of distributing the work to one CPU core at a time with the basic for loop, a parallel for loop can distribute the work to all available cores in parallel. This results in a speedup close to the number of CPU cores.

```
// single-core
for (int x = 0; x < 1000; x++) { /* do stuff */ }
// multi-core
Parallel.For(0, 1000, x => { /* do stuff */ });
```

After that we can use span arrays in C#, which gives us many advantages.

```
// Span<float> default
Span<float> spanArray1 = new float[1000];
// Span<float> from float[] array
float[] floatArray = new float[1000];
Span<float> spanArray2 = floatArray.AsSpan();
// ReadOnlySpan<float> using stackalloc
ReadOnlySpan<float> spanArray3 = stackalloc float[1000];
// float[] pass as Span<float> and return
float[] floatArray2 = new float[1000];
var spanArray4 = ReturnAsSpan(floatArray2);
static Span<float> ReturnAsSpan(Span<float> array) => array;
```

SIMD can be used with span arrays. SIMD can do one step for multiple data.

```
// SIMD Vector512, c# 12
var part1 = spanArray1.Slice(900, 100);
Span<Vector512<float>> vector1 = MemoryMarshal.Cast<float, Vector512<float>>(part1);
for (int v = 0; v < vector1.Length; v++) // SIMD
    vector1[v] = Vector512.Max(vector1[v], Vector512<float>.Zero);
for (int k = vector1.Length * Vector512<float>.Count; k < part1.Length; k++)
    part1[k] = MathF.Max(part1[k], 0); // remaining elements

// SIMD Vector, Vector128, Vector256
var part2 = floatArray.AsSpan().Slice(900, 100);
var vector2 = MemoryMarshal.Cast<float, Vector256<float>>(part2);
for (int v = 0; v < vector2.Length; v++) // SIMD
    vector2[v] += vector2[v];
for (int k = vector2.Length * Vector256<float>.Count; k < part2.Length; k++)
    part2[k] += part2[k]; // remaining elements
```

Parallel + Span + SIMD = speedup 20x on my 5600X CPU. Now let's push forward.

Code Concept

I really like top-level statement code, static functions, and local functions in C#. It keeps the code clean and focused on the important parts. My goal is to put everything in one file when I create a demo. I tried to find a good balance between the presentation and the basic demo for NN and CNN.

Pre-Activation Concept

The usual way to compute a linear layer is to compute the dot product, then add the bias weight, and then activate the accumulated sum. We can call it inputs to output. My implementation omits the bias weight and goes the other way, input to outputs. Pre-activation means to use ReLU as activation function for input neurons to avoid calculations with zero values. This can boost the NN to run ~5 times faster.

Conventions

Forget them all when you try to do such a project yourself. You have to test, you have to test everything as best you can. And you have to know that parallel computing will lead to non-reproducible CNN training results in an erratic way.

Code conventions can be A, then B, then C. But my code will do it in every way I have found that works better. Sometimes B then A can increase speed by more than 20%. But if we stack enough bad decisions in the runtime, we can lose all the benefits we have gained so far.

Forget conventions also means that there is no single right way to build a CNN. Maybe a loose idea that has changed over time. Follow what works best.

The CNN

Google about convolution: “A thing that is complex and difficult to follow: convolution”

Forget that! We fit the same image into the CNN as we did with the NN. The architecture of the CNN then determines the size of the output layer, which acts as the input layer of the NN.

Instead of adding an input neuron times its weight to an output neuron. The CNN way is to add the input map times its kernel, which can consist of many weights, to the output map.

```
// define cnn
int inputDimension = 28; // sqrt(784)
int[] cnn = { 1, 8, 24 }; // non-RGB = 1 (MNIST), 3 = RGB
int[] filter = { 6, 6 }; // x * y kernel dim
int[] stride = { 1, 3 }; // replaces pooling with higher strides than 1
```

This is how we define the convolutional neural network. The first layer is the input layer with one dimension (`cnn[0]`) of a 2-dimensional grayscale image with $28 * 28 = 784$ pixels from the MNIST data. We multiply the image with a 6x6 (`filter[0]`) kernel filter consisting of 36 weights and a stride step size of 1 (`stride[0]`) and create 8 (`cnn[1]`) output maps with 8 (`filter[0]`) different kernel filters with the dimension $23 * 23 = 529$ for each output map. The first hidden layer of the CNN has 4232 neurons in total.

These 8 (`cnn[1]`) maps with 529 neurons are the input maps for the next layer. Where we use $8 * 24$ kernels with 6x6 (`filter[1]`) = 192 kernels. This means we stack 8 input maps with 8 kernels and a stride of 3 (`stride[1]`) onto each of the 24 (`cnn[2]`) output maps. The result is the CNN output layer with 24 maps with dimension $6 * 6 = 36$ neurons on each map. The CNN output layer has 864 neurons. Which act as input neurons for the NN.

The normal way could be to define everything by hand. This implementation uses arrays (`cnn`, `filter`, `stride`) to define the CNN over many layers. There is a lot to learn about how all the complicated interactions work, and creating and defining a CNN is much more challenging than an NN for many reasons. These methods: “`CnnDimensions()`”, “`CnnSteps()`”, “`KernelSteps()`” help us to run through the network structure of the CNN allowing us to achieve forward and backward more easily. But the concept in its essence stays the same as we did with the NN.

The best way to understand the idea is to understand the code and run it. You could start with only one CNN layer with 2 output maps and a 3x3 kernel with stride 1, then you increase the architecture with more maps and layers and play around to get a feeling.

My fine-tuned CNN demo was the result of a lot of trial and error. What worked best was the direction to go. So it is not a classical CNN, that's for sure. No Padding, no Pooling, no Bias, no Kernel-Flip, no Sigmoid. The code is optimized to run fast. Also the neural net functions are highly optimized for this CNN.

Template CNN Forward Pass

```
/* for (int i = 0; i < cnn.Length - 1; i++) // 1. start to end
{
    int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1], // 2. layer steps
    lStep = cSteps[i], rStep = cSteps[i + 1], kd = filter[i], ks = kStep[i],
    st = stride[i], lMap = lDim * lDim, rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;

    for (int l = 0, ls = lStep; l < left; l++, ls += lMap) // input channel map
        for (int r = 0, rs = rStep; r < right; r++, rs += rMap) // output channel map
            for (int y = 0, k = rs, w = ks + (l * right + r) * kMap; y < rDim; y++) // conv dy
                for (int x = 0; x < rDim; x++, k++) // conv dim x
                {
                    int j = ls + y * sDim + x * st; // input map position
                    for (int col = 0, fid = 0; col < kd; col++) // filter dim y
                        for (int row = 0; row < col * lDim + kd; row++, fid++) // filter dim x
                        {
                            // do stuff
                        }
                }
} */
```

Template CNN Backward Pass

```
/* for (int i = cnn.Length - 2; i >= 0; i--) // 1. end to start
{
    int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1], // 2. layer steps
    lStep = cSteps[i], rStep = cSteps[i + 1], kd = filter[i], ks = kStep[i],
    st = stride[i], lMap = lDim * lDim, rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;

    for (int l = 0, ls = lStep; l < left; l++, ls += lMap) // input channel map
        for (int r = 0, rs = rStep; r < right; r++, rs += rMap) // output channel map
            for (int y = 0, k = rs, w = ks + (l * right + r) * kMap; y < rDim; y++) // conv dy
                for (int x = 0; x < rDim; x++, k++) // conv dim x
                {
                    int j = ls + y * sDim + x * st; // input map position
                    for (int col = 0, fid = 0; col < kd; col++) // filter dim y cols
                        for (int row = 0; row < col * lDim + kd; row++, fid++) // fdim x rows
                        {
                            // do stuff
                        }
                }
} */
```

CNN Forward Basic

```
static void ConvForwardBasic(int[] cnn, int[] dim, int[] cs, int[] filter, int[] kStep, int[]
stride, Span<float> conv, float[] kernel)
{
    for (int i = 0; i < cnn.Length - 1; i++)
    {
        int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1],
        lStep = cs[i], rStep = cs[i + 1], kd = filter[i], ks = kStep[i], st = stride[i],
        lMap = lDim * lDim, rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;

        // convolution
        for (int l = 0, ls = lStep; l < left; l++, ls += lMap) // input channel map
            for (int r = 0, rs = rStep; r < right; r++, rs += rMap) // output channel map
            {
                int k = rs; // output map position
                for (int y = 0, w = ks + (l * right + r) * kMap; y < rDim; y++) // conv dim y
                    for (int x = 0; x < rDim; x++, k++) // conv dim x
                    {
                        float sum = 0;
                        int j = ls + y * sDim + x * st; // input map position
                        for (int col = 0, fid = 0; col < kd; col++) // filter dim y then x
                            for (int row = col * lDim; row < col * lDim + kd; row++, fid++)
                                sum += conv[j + row] * kernel[w + fid];
                        conv[k] += sum;
                    }
            }

        // relu activation for each output conv map
        for (int k = rStep; k < rStep + rMap * right; k++)
            conv[k] = conv[k] > 0 ? conv[k] : 0; // relu
    }
}
```

CNN Backward Basic

```
static void ConvBackpropBasic(
int[] cnn, int[] dim, int[] cSteps, int[] filter, int[] kStep, int[] stride, Span<float>
kernel, Span<float> kernel_delta, Span<float> cnnGradient, Span<float> conv)
{
    // one loop bp: convolution gradient and kernel delta
    for (int i = cnn.Length - 2; i >= 0; i--)
    {
        int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1],
        lStep = cSteps[i], rStep = cSteps[i + 1], kd = filter[i],
        ks = kStep[i], st = stride[i], lMap = lDim * lDim,
        rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;

        for (int l = 0, ls = lStep; l < left; l++, ls += lMap) // input channel map
            for (int r = 0, rs = rStep; r < right; r++, rs += rMap) // output channel map
                for (int y = 0, k = rs, w = ks + (1 * right + r) * kMap; y < rDim; y++) // cdy
                    for (int x = 0; x < rDim; x++, k++) // conv dim x
                        if (conv[k] > 0) // relu derivative
                        {
                            float gra = cnnGradient[k];
                            int j = ls + y * sDim + x * st; // input map position
                            for (int col = 0, fid = 0; col < kd; col++) // filter dim y cols
                                for (int row = col * lDim; row < col * lDim + kd; row++, fid++)
                                    { // filter dim x rows
                                        cnnGradient[j + row] = kernel[w + fid] * gra;
                                        kernel_delta[w + fid] += conv[j + row] * gra;
                                    }
                        }
    }
}
```

The demo is somewhere between basic and advanced, but works pretty good. Now I'm proud to present my CNN demo 2024, you can run it the same way as the NN, just keep the `struct AutoData{...}` and copy the following code.

```
// .NET 8
// copy code
// then press to close functions: (ctrl + m + o)
// use Release mode, Debug is slow!
```



```

using System.Diagnostics; using System.IO.Compression;
using System.Runtime.InteropServices; using System.Runtime.Intrinsics;

// 0. load MNIST data
Console.WriteLine("\nBegin convolutional neural network demo\n");
AutoData d = new(@"C:\cnn2024\", AutoData.Dataset.MNIST); // get data
// 1. init cnn + nn + hyperparameters
int[] cnn = { 1, 8, 24 }; // input layer: non-RGB = 1, RGB = 3
int[] filter = { 6, 6 }; // x * y dim for kernel
int[] stride = { 1, 3 }; // pooling with higher strides than 1
int[] net = { 784, 300, 300, 10 }; // nn
var LR = 0.005f;
var MOMENTUM = 0.5f;
var DROPOUT = 0.5f;
var SEED = 274024;
var FACTOR = 0.95f;
int BATCH = 42;
int EPOCHS = 50;

// 2.0 convolution dimensions
int[] dim = CnnDimensions(cnn.Length - 1, 28, filter, stride);
// 2.1 convolution steps for layerwise preparation
int[] cStep = CnnSteps(cnn, dim);
// 2.2 kernel steps for layerwise preparation
int[] kStep = KernelSteps(cnn, filter);
// 2.3 init visual based kernel weights
float[] kernel = InitConvKernel(cnn, filter, SEED);
// 2.4 init neural network weights w.r.t. cnn
float[] weights = InitNeuralNetWeights(net, cStep, SEED);

NetInfo();
Random rng = new(SEED); Stopwatch sw = Stopwatch.StartNew();
int[] indices = Enumerable.Range(0, 60000).ToArray(), done = new int[60000];
// 4.0
Console.WriteLine("\nStarting training");
float[] delta = new float[weights.Length], kDelta = new float[kernel.Length];
for (int epoch = 0; epoch < EPOCHS; epoch++, LR *= FACTOR, MOMENTUM *= FACTOR)
    CnnTraining(true, indices, d.samplesTrainingF, d.labelsTraining, rng.Next(),
        done, cnn, filter, stride, kernel, kDelta, dim, cStep, kStep, net,
        weights, delta, 60000, epoch, BATCH, LR, MOMENTUM, DROPOUT);
Console.WriteLine($"Done after {(sw.Elapsed.TotalMilliseconds / 1000.0):F2}s\n");
// 5.0
CnnTesting(d.samplesTestF, d.labelsTest, cnn, filter, stride, kernel, dim, cStep, kStep, net,
    weights, 10000); Console.WriteLine("\nEnd CNN demo");

```

```

// 2.0 conv dimensions
static int[] CnnDimensions(int cnn_layerLen, int startDimension, int[] filter, int[] stride)
{
    int[] dim = new int[cnn_layerLen + 1];
    for (int i = 0, c_dim = (dim[0] = startDimension); i < cnn_layerLen; i++)
        dim[i + 1] = c_dim = (c_dim - (filter[i] - 1)) / stride[i];
    return dim;
}

// 2.1 convolution steps
static int[] CnnSteps(int[] cnn, int[] dim)
{
    int[] cs = new int[cnn.Length + 1];
    cs[1] = dim[0] * dim[0]; // startDimension^2
    for (int i = 0, sum = cs[1]; i < cnn.Length - 1; i++)
        cs[i + 2] = sum += cnn[i + 1] * dim[i + 1] * dim[i + 1];
    return cs;
}

// 2.2 kernel steps in structure for kernel weights
static int[] KernelSteps(int[] cnn, int[] filter)
{
    int[] ks = new int[cnn.Length - 1];
    for (int i = 0; i < cnn.Length - 2; i++)
        ks[i + 1] += cnn[i + 0] * cnn[i + 1] * filter[i] * filter[i];
    return ks;
}

// 2.3 init kernel weights
static float[] InitConvKernel(int[] cnn, int[] filter, int seed)
{
    int cnn_weightLen = 0;
    for (int i = 0; i < cnn.Length - 1; i++)
        cnn_weightLen += cnn[i] * cnn[i + 1] * filter[i] * filter[i];
    float[] kernel = new float[cnn_weightLen]; Random rnd = new(seed);
    for (int i = 0, c = 0; i < cnn.Length - 1; i++) // each cnn layer
        for (int l = 0, f = filter[i]; l < cnn[i]; l++) // each input map
            for (int r = 0; r < cnn[i + 1]; r++) // each output map
                for (int col = 0; col < f; col++) // kernel y
                    for (int row = 0; row < f; row++, c++) // kernel x
                        kernel[c] = (rnd.NextSingle() * 2 - 1) /
                            MathF.Sqrt(cnn[i] * cnn[i + 1] * 0.5f);
    return kernel;
}

```

```

// 2.4 init neural network weights for cnn
static float[] InitNeuralNetWeights(int[] net, int[] convMapsStep, int seed)
{
    // 3.1.1 fit cnn output to nn input
    net[0] = convMapsStep[1] - convMapsStep[2];
    // 3.1.2 glorot nn weights init
    int len = 0;
    for (int n = 0; n < net.Length - 1; n++)
        len += net[n] * net[n + 1];

    float[] weight = new float[len];
    Random rnd = new(seed);
    for (int i = 0, m = 0; i < net.Length - 1; i++, m += net[i] * net[i + 1]) // layer
        for (int w = m; w < m + net[i] * net[i + 1]; w++) // weights
            weight[w] = (rnd.NextSingle() * 2 - 1)
                * MathF.Sqrt(6.0f / (net[i] + net[i + 1])) * 0.5f;
    return weight;
}

// 2.5 cnn number of neurons
static int CnnNeuronsLen(int startDimension, int[] cnn, int[] dim)
{
    int cnn_layerLen = cnn.Length - 1;
    int cnn_neuronLen = startDimension * startDimension; // add input first
    for (int i = 0; i < cnn_layerLen; i++)
        cnn_neuronLen += cnn[i + 1] * dim[i + 1] * dim[i + 1];
    return cnn_neuronLen;
}

// 2.6 nn number of neurons
static int NeuronsLen(int[] net)
{
    int sum = 0;
    for (int n = 0; n < net.Length; n++)
        sum += net[n];
    return sum;
}

```

```

// 3.0 cnn ff
static void ConvForward(int[] cnn, int[] dim, int[] cs, int[] filter,
    int[] kStep, int[] stride, Span<float> conv, Span<float> kernel)
{
    for (int i = 0; i < cnn.Length - 1; i++)
    {
        int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1],
        lStep = cs[i + 0], rStep = cs[i + 1], kd = filter[i], ks = kStep[i], st = stride[i],
        lMap = lDim * lDim, rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;
        // better cache locality?
        for (int l = 0, leftStep = lStep; l < left; l++, leftStep += lMap) // input map
        {
            var inpMap = conv.Slice(leftStep, lDim * lDim);
            for (int r = 0, rightStep = rStep; r < right; r++, rightStep += rMap) // out map
            {
                var outMap = conv.Slice(rightStep, rDim * rDim);
                for (int col = 0; col < kd; col++) // kernel filter dim y
                {
                    var kernelRow = kernel.Slice(ks + (l * right + r) * kMap + kd * col, kd);
                    for (int y = 0, kk = 0; y < rDim; y++) // conv dim y
                    {
                        var inputRow = inpMap.Slice((y * st + col) * lDim, lDim);
                        for (int x = 0; x < rDim; x++, kk++) // conv dim x
                        {
                            float sum = 0; // kernel filter dim x
                            for (int row = 0, rowStep = x * st; row < kernelRow.Length; row++)
                                sum = kernelRow[row] * inputRow[rowStep + row] + sum;
                            outMap[x + y * rDim] += sum;
                        }
                    }
                }
            }
        }
        // relu activation
        var activateMaps = conv.Slice(rStep, rDim * rDim * right);
        var vec = MemoryMarshal.Cast<float>, Vector512<float>>(activateMaps);
        for (int v = 0; v < vec.Length; v++) // SIMD
            vec[v] = Vector512.Max(vec[v], Vector512<float>.Zero);
        for (int k = vec.Length * Vector512<float>.Count; k < activateMaps.Length; k++) // cm
            activateMaps[k] = MathF.Max(activateMaps[k], 0);
    }
}

```

```

// 3.1 dropout between cnn output and nn input
static void Dropout(int seed, Span<float> conv, int start, int part, float drop)
{
    Random rng = new(seed);
    var outMap = conv.Slice(start, part);
    for (int k = 0; k < outMap.Length; k++) // conv map
    {
        float sum = outMap[k];
        if (sum > 0)
            outMap[k] = rng.NextSingle() > drop ? sum : 0;
    }
}

// 3.2 nn ff
static void FeedForward(Span<int> net, Span<float> weights, Span<float> neurons)
{
    for (int i = 0, k = net[0], w = 0; i < net.Length - 1; i++) // layers
    {
        // slice input + output layer
        var outLocal = neurons.Slice(k, net[i + 1]);
        var inpLocal = neurons.Slice(k - net[i], net[i]);
        for (int l = 0; l < inpLocal.Length; l++, w = outLocal.Length + w) // input neurons
        {
            // fast input neuron
            var inpNeuron = inpLocal[l];
            if (inpNeuron <= 0) continue; // ReLU input pre-activation
            // slice connected weights
            var wts = weights.Slice(w, outLocal.Length);
            // span to vector
            var wtsVec = MemoryMarshal.Cast<float, Vector512<float>>(wts);
            var outVec = MemoryMarshal.Cast<float, Vector512<float>>(outLocal);
            // SIMD
            for (int v = 0; v < outVec.Length; v++)
                outVec[v] = wtsVec[v] * inpNeuron + outVec[v];
            // compute remaining output neurons
            for (int r = wtsVec.Length * Vector512<float>.Count; r < outLocal.Length; r++)
                outLocal[r] = wts[r] * inpNeuron + outLocal[r];
        }
        k = outLocal.Length + k; // stack output id
    }
}

```

```

// 3.3 softmax
static int SoftArgMax(Span<float> neurons)
{
    int id = 0; // argmax
    float max = neurons[0];
    for (int i = 1; i < neurons.Length; i++)
        if (neurons[i] > max) { max = neurons[i]; id = i; }
    // softmax activation
    float scale = 0;
    for (int n = 0; n < neurons.Length; n++)
        scale += neurons[n] = MathF.Exp((neurons[n] - max));
    for (int n = 0; n < neurons.Length; n++)
        neurons[n] /= scale; // pseudo probabilities
    return id; // return nn prediction
}

// 3.4 output error gradient (target - output)
static void ErrorGradient(Span<float> neurons, int target)
{
    for (int i = 0; i < neurons.Length; i++)
        neurons[i] = target == i ? 1 - neurons[i] : -neurons[i];
}

```

```

// 3.5 nn bp
static void Backprop(Span<float> neurons, Span<int> net, Span<float> weights,
    Span<float> deltas)
{
    int j = neurons.Length - net[^1], k = neurons.Length, m = weights.Length;
    for (int i = net.Length - 2; i >= 0; i--)
    {
        int right = net[i + 1], left = net[i];
        k -= right; j -= left; m -= right * left;
        // slice input + output layer
        var inputNeurons = neurons.Slice(j, left);
        var outputGradients = neurons.Slice(k, right);
        for (int l = 0, w = m; l < left; l++, w += right)
        {
            var n = inputNeurons[l];
            if (n <= 0) { inputNeurons[l] = 0; continue; }
            // slice connected weights + deltas
            var wts = weights.Slice(w, right); // var inVec = Vector256.Create(n);
            var dts = deltas.Slice(w, right);
            // turn to vector
            var wtsVec = MemoryMarshal.Cast<float, Vector256<float>>(wts);
            var dtsVec = MemoryMarshal.Cast<float, Vector256<float>>(dts);
            var graVec = MemoryMarshal.Cast<float, Vector256<float>>(outputGradients);
            var sumVec = Vector256<float>.Zero;
            for (int v = 0; v < graVec.Length; v++) // SIMD, gradient sum and delta
            {
                var outGraVec = graVec[v];
                sumVec = wtsVec[v] * outGraVec + sumVec;
                dtsVec[v] = n * outGraVec + dtsVec[v];
            }
            // turn vector sum to float
            var sum = Vector256.Sum(sumVec);
            // compute remaining elements
            for (int r = graVec.Length * Vector256<float>.Count; r < wts.Length; r++)
            {
                var outGraSpan = outputGradients[r];
                sum = wts[r] * outGraSpan + sum;
                dts[r] = n * outGraSpan + dts[r];
            }
            inputNeurons[l] = sum; // reuse for gradients now
        }
    }
}

```

```

// 3.6 cnn bp
static void ConvBackprop(int[] cnn, int[] dim, int[] cSteps, int[] filter, int[] kStep, int[]
stride, Span<float> kernel, Span<float> kDelta, Span<float> cnnGradient, Span<float> conv)
{
    for (int i = cnn.Length - 2; i >= 0; i--) // one loop bp: cnn gradient and kernel delta
    {
        int left = cnn[i], right = cnn[i + 1], lDim = dim[i], rDim = dim[i + 1],
        lStep = cSteps[i], rStep = cSteps[i + 1], kd = filter[i], ks = kStep[i], st =
        stride[i], lMap = lDim * lDim, rMap = rDim * rDim, kMap = kd * kd, sDim = st * lDim;

        for (int l = 0; l < left; l++, lStep += lMap) // input channel map
        {
            var inpMap = conv.Slice(lStep, lMap);
            var inpGraMap = cnnGradient.Slice(lStep, lMap);
            for (int r = 0, rs = rStep; r < right; r++, rs += rMap) // output channel map
            {
                var outMap = conv.Slice(rs, rMap);
                var graMap = cnnGradient.Slice(rs, rMap);
                for (int col = 0; col < kd; col++) // filter dim y cols
                {
                    int kernelID = ks + (l * right + r) * kMap + kd * col;
                    var kernelRow = kernel.Slice(kernelID, kd);
                    var kernelDeltaRow = kDelta.Slice(kernelID, kd);
                    for (int y = 0; y < rDim; y++) // conv dim y
                    {
                        int irStep = (y * st + col) * lDim;
                        var inputRow = inpMap.Slice(irStep, lDim);
                        var inputGraRow = inpGraMap.Slice(irStep, lDim);
                        int outStep = y * rDim;
                        for (int x = 0; x < rDim; x++) // conv dim x
                            if (outMap[x + outStep] > 0) // relu derivative
                            {
                                float gra = graMap[x + outStep];
                                for (int row = 0, rowStep = x * st; row < kd; row++) // fdx rw
                                {
                                    kernelDeltaRow[row] += inputRow[rowStep + row] * gra;
                                    inputGraRow[rowStep + row] += kernelRow[row] * gra;
                                }
                            }
                    }
                }
            }
        }
    }
}

```



```

// 3.7 sgd
static void Update(Span<float> weights, Span<float> delta, float lr, float mom)
{
    var weightVecArray = MemoryMarshal.Cast<float, Vector512<float>>(weights);
    var deltaVecArray = MemoryMarshal.Cast<float, Vector512<float>>(delta);
    // SIMD
    for (int v = 0; v < weightVecArray.Length; v++)
    {
        weightVecArray[v] = deltaVecArray[v] * lr + weightVecArray[v];
        deltaVecArray[v] *= mom;
    }
    // remaining elements
    for (int w = weightVecArray.Length * Vector512<float>.Count; w < weights.Length; w++)
    {
        weights[w] = delta[w] * lr + weights[w];
        delta[w] *= mom;
    }
}

```

```

// 4.0 train sample
static int TrainSample(int id, int target, int seedD, int[] done, float[] data,
int[] cnn, int[] dim, int[] filter, int[] stride, float[] kernel, float[] kernelDelta,
int[] cSteps, int[] kStep, int[] net, float[] weight, float[] delta,
int cnnLen, int nnLen, float drop)
{
    if (done[id] >= 5) return -1; // drop easy examples
    // feed conv input layer with sample
    Span<float> conv = new float[cnnLen];
    data.AsSpan().Slice(id * 784, 784).CopyTo(conv);
    // conv feed forward + dropout (cnn output == nn input) layer
    ConvForward(cnn, dim, cSteps, filter, kStep, stride, conv, kernel);
    int right = cnn[^1], rDim = dim[^1], start = cSteps[^2];
    Dropout(seedD, conv, start, rDim * rDim * right, drop);
    // neural net feed forward
    Span<float> neuron = new float[nnLen];
    conv.Slice(cSteps[^2], net[0]).CopyTo(neuron);
    FeedForward(net, weight, neuron);
    var outs = neuron.Slice(nnLen - net[^1], net[^1]);
    int prediction = SoftArgMax(outs);
    // drop easy examples
    if (outs[target] >= 0.9999) { done[id] += 1; return -1; }
    if (outs[target] >= 0.9) return prediction; // probability check
    // neural net backprop
    ErrorGradient(outs, target);
    Backprop(neuron, net, weight, delta);
    // conv net backprop
    Span<float> cnnGradient = new float[cSteps[^2] + net[0]];
    neuron.Slice(0, net[0]).CopyTo(cnnGradient.Slice(cSteps[^2], net[0]));
    ConvBackprop(cnn, dim, cSteps, filter, kStep,
stride, kernel, kernelDelta, cnnGradient, conv);
    return prediction;
}

```

```

// 5.0 train epoch
static float CnnTraining(bool multiCPU, int[] indices, float[] data, byte[] label,
    int seed, int[] done, int[] cnn, int[] filter, int[] stride, float[] kernel,
    float[] kernelDelta, int[] dim, int[] cSteps, int[] kStep, int[] net, float[] weight,
    float[] delta, int len, int epoch, int batch, float lr, float mom, float drop)
{
    DateTime elapsed = DateTime.Now; Random.Shared.Shuffle(indices); // shuffle ids
    int correct = 0, all = 0; Random rng = new Random(seed + epoch);
    int cnnLen = CnnNeuronsLen(28, cnn, dim), nnLen = NeuronsLen(net);
    // batch training
    for (int b = 0, B = (int)(len / batch); b < B; b++) // each batch for one epoch
    {
        int[] rand = Enumerable.Repeat(0, batch).Select(_ => rng.Next()).ToArray();
        if (multiCPU) Parallel.For(0, batch, x => // each sample in this batch
        {
            // get shuffled supervised sample id
            int id = indices[x + b * batch], target = label[id];
            int prediction = TrainSample(id, target, rand[x], done, data,
                cnn, dim, filter, stride, kernel, kernelDelta, cSteps,
                kStep, net, weight, delta, cnnLen, nnLen, drop);
            if (prediction != -1)
            {
                if (prediction == target) Interlocked.Increment(ref correct);
                Interlocked.Increment(ref all); // statistics accuracy
            }
        });
        else for (int x = 0; x < batch; x++)
        {
            int id = indices[x + b * batch], target = label[id];
            int prediction = TrainSample(id, target, rand[x], done, data,
                cnn, dim, filter, stride, kernel, kernelDelta, cSteps,
                kStep, net, weight, delta, cnnLen, nnLen, drop);
            if (prediction != -1) {
                if (prediction == target) Interlocked.Increment(ref correct);
                Interlocked.Increment(ref all); // statistics accuracy
            }
        }

        Update(kernel, kernelDelta, lr, 0); // no cnn mom works better?
        Update(weight, delta, lr, mom);
    }
    if ((epoch + 1) % 10 == 0)
        Console.WriteLine($"epoch = {(epoch + 1):00} | acc = {(correct * 100.0 / all):F2}%"+
            + $" | time = {(DateTime.Now - elapsed).TotalSeconds:F2}s");
    return (correct * 100.0f / all);
}

```

```

// 6.0 test cnn on unseen data
static void CnnTesting(float[] data, byte[] label, int[] cnn, int[] filter, int[] stride,
float[] kernel, int[] dim, int[] cSteps, int[] kStep, int[] net, float[] weight, int len)
{
    DateTime elapsed = DateTime.Now;
    // cnn stuff
    int cnnLen = CnnNeuronsLen(28, cnn, dim), nnLen = NeuronsLen(net);
    // correction value for each neural network weight
    int correct = 0;
    Parallel.For(0, len, id =>
    {
        int target = label[id];
        Span<float> conv = new float[cnnLen];
        data.AsSpan().Slice(id * 784, 784).CopyTo(conv);
        // convolution feed forward
        ConvForward(cnn, dim, cSteps, filter, kStep, stride, conv, kernel);
        // copy cnn output to nn input
        Span<float> neuron = new float[nnLen];
        conv.Slice(cSteps[^2], net[0]).CopyTo(neuron);
        // neural net feed forward
        FeedForward(net, weight, neuron);
        int prediction = SoftArgMax(neuron.Slice(nnLen - net[^1], net[^1]));
        // statistics accuracy
        if (prediction == target)
            Interlocked.Increment(ref correct);
    });
    Console.WriteLine($"Test accuracy = {(correct * 100.0 / len):F2}% " +
        $" after {(DateTime.Now - elapsed).TotalSeconds:F2}s");
}

```

```

void NetInfo()
{
    #if DEBUG
    Console.WriteLine("Debug mode is on, switch to Release mode"); return;
    #endif

    Console.WriteLine($"Convolution = {string.Join("-", cnn)}");
    Console.WriteLine($"Kernel size = {string.Join("-", filter)}");
    Console.WriteLine($"Stride step = {string.Join("-", stride)}");
    Console.WriteLine($"DimensionX{" ",2}= {string.Join("-", dim)}");
    Console.WriteLine($"Map (Dim²){" ",2}= {string.Join("-", dim.Select(x => x * x))}");
    Console.WriteLine($"CNN+NN{" ",6}= " +
        $" {string.Join("-", cnn.Zip(dim, (x, d) => x * d * d))}+{string.Join("-", net)}");
    Console.WriteLine($"CNN weights{" ",1}= {kernel.Length} ({cnn.Zip(cnn.Skip(1),
        (p, n) => p * n).Sum()})");
    Console.WriteLine($"NN weights{" ",2}= {weights.Length}");
    Console.WriteLine($"SEED{" ",8}= {SEED:F0}");
    Console.WriteLine($"EPOCHS{" ",6}= {EPOCHS:F0}");
    Console.WriteLine($"BATCHSIZE{" ",3}= {BATCH:F0}");
    Console.WriteLine($"CNN LR{" ",6}= {LR:F3} | MLT = {FACTOR:F2}");
    Console.WriteLine($"NN LR{" ",7}= {LR:F3} | MLT = {FACTOR:F2}");
    Console.WriteLine($"Momentum{" ", 4}= {MOMENTUM:F2}{" ", 2}| MLT = {FACTOR:F2}");
    Console.WriteLine($"Dropout{" ", 5}= {DROPOUT:F2}");
}

// next steps could be:
// 7.0 save cnn+nn
// 8.0 load cnn+nn

```

```

Begin convolutional neural network demo

Dataset: MNIST (C:\cnn2024\))

Convolution = 1-8-24
Kernel size = 6-6
Stride step = 1-3
DimensionX = 28-23-6
Map (Dim2) = 784-529-36
CNN+NN = 784-4232-864+864-300-300-10
CNN weights = 7200 (200)
NN weights = 352200
SEED = 274024
EPOCHS = 50
BATCHSIZE = 42
CNN LR = 0.005 | MLT = 0.95
NN LR = 0.005 | MLT = 0.95
Momentum = 0.50 | MLT = 0.95
Dropout = 0.50

Starting training
epoch = 10 | acc = 98.31% | time = 2.66s
epoch = 20 | acc = 98.58% | time = 1.74s
epoch = 30 | acc = 98.59% | time = 1.26s
epoch = 40 | acc = 98.73% | time = 0.92s
epoch = 50 | acc = 98.62% | time = 0.76s
Done after 86.10s




















Test accuracy = 99.52% after 0.41s

End CNN demo

```

Trained in under 2 minutes, the demo result of 99.52% accuracy was one of the best on many runs, but may vary on your machine. In other words, out of 10,000 unknown test examples, the network was able to predict 9,952 correctly and failed on 48 examples.

Leaderboard MNIST (21-40)

21	Second Order Neural Ordinary Differential Equation	0.37	99.63	×	On Second Order Behaviour in Augmented Neural ODEs			2020
22	Augmented Neural Ordinary Differential Equation	0.37	99.63	×	Augmented Neural ODEs			2019
23	Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis	0.4		×				
24	Energy-Based Sparse Representation	0.4		×				
25	DBSNN	0.4		×				
26	DSN	0.4		×	Deeply-Supervised Nets			2014
27	CKN	0.4		×	Convolutional Kernel Networks			2014
28	C-SVDDNet	0.4		×	Unsupervised Feature Learning with C-SVDDNet			2014
29	HOPE	0.4		×	Hybrid Orthogonal Projection and Estimation (HOPE): A New Framework to Probe and Learn Neural Networks			2015
30	FLSCNN	0.4		×	Enhanced Image Classification With a Fast-Learning Shallow Convolutional Neural Network			2015
31	MLR DNN	0.4		×				
32	MIM	0.4		×	On the Importance of Normalisation Layers in Deep Learning with Piecewise Linear Activation Units			2015
33	Fitnet-LSUV-SVM	0.4		×	All you need is a good init			2015
34	Deformation Models	0.5		×				
35	Trainable feature extractor	0.5		×				
36	The Best Multi-Stage Architecture	0.5		×				
37	COSFIRE	0.5		×				
38	Maxout Networks	0.5		×	Maxout Networks			2013
39	NiN	0.5		×	Network In Network			2013
40	ReNet	0.5		×	ReNet: A Recurrent Neural Network Based Alternative to Convolutional Networks			2015

[MNIST Benchmark \(Image Classification\) | Papers With Code](#)

Even in competition is this CNN close to a top performer, rank 34 is nice. The best thing is that we have a really clean implementation that can give such a prediction quality. Further techniques, especially data augmentation techniques like rotation, scaling and shifting, can easily improve our model.

```

Begin fast convolutional neural network demo

Dataset: FashionMNIST (C:\fast_cnn\)

Convolution = 1-12-28
Kernel size = 3-5
Stride step = 1-2
DimensionX = 28-26-11
Map (Dim²) = 784-676-121
CNN = 784-8112-3388
NN = 3388-400-400-10
Kernel weights = 8508
Network weights = 1519200
Learning = 0.00600 | MLT = 0.95
Momentum = 0.50 | MLT = 0.95
Infinity = 0.50

Starting training
epoch = 10 | acc = 90.15% | time = 6.42s
Accuracy on test data = 90.60% after 0.86s
epoch = 20 | acc = 92.85% | time = 5.62s
Accuracy on test data = 91.66% after 0.87s
epoch = 30 | acc = 94.69% | time = 5.43s
Accuracy on test data = 92.09% after 0.86s
epoch = 40 | acc = 95.52% | time = 5.09s
Accuracy on test data = 92.16% after 0.93s
epoch = 50 | acc = 95.89% | time = 5.11s
Accuracy on test data = 92.43% after 0.88s
epoch = 60 | acc = 96.16% | time = 4.86s
Accuracy on test data = 92.30% after 0.90s
epoch = 70 | acc = 96.36% | time = 4.81s
Accuracy on test data = 92.39% after 0.92s
epoch = 80 | acc = 96.33% | time = 4.74s
Accuracy on test data = 92.46% after 0.94s
epoch = 90 | acc = 96.21% | time = 4.60s
Accuracy on test data = 92.35% after 1.00s
epoch = 100 | acc = 96.38% | time = 4.62s
Accuracy on test data = 92.45% after 0.95s
Done after 538.67s

Accuracy on test data = 92.45% after 0.86s

Load CNN again and test it
Accuracy on test data = 92.45% after 1.00s

End MNIST CNN demo


















```

Fashion-MNIST is the next step and is known as a lot harder than MNIST. Fine-tuning plus more training led to 92.45% test accuracy.

To run this dataset change:

```
AutoData d = new(@"C:\cnn2024\", AutoData.Dataset.FashionMNIST);
```


Leaderboard Fashion-MNIST (11-20)

11	StiDi-BP in R-CSNN	7.2	92.8	Spike time displacement based error backpropagation in convolutional spiking neural networks			2021	SNN
12	NeuPDE	7.6		NeuPDE: Neural Network Based Ordinary and Partial Differential Equations for Modeling Time-Dependent Data			2019	
13	Star Algorithm on LeNet	7.7	92.3	Star algorithm for NN ensembling			2022	
14	Convolutional Tsetlin Machine	8.6	91.4	The Convolutional Tsetlin Machine			2019	
15	OTTT	9.6		Online Training Through Time for Spiking Neural Networks			2022	SNN
16	GECCO	11.91	88.09	A Single Graph Convolution Is All You Need: Efficient Grayscale Image Classification			2024	
17	pFedBreD_ns_mg		99.06	Personalized Federated Learning with Hidden Information on Personalized Prior			2022	
18	FastSNN (CNN)		90.57	Robust and accelerated single-spike spiking neural network training with applicability to challenging temporal tasks			2022	SNN
19	FastSNN (MLP)		89.05	Robust and accelerated single-spike spiking neural network training with applicability to challenging temporal tasks			2022	SNN
20	Sparse Spiking Gradient Descent (CNN)		86.7	Sparse Spiking Gradient Descent			2021	

[Fashion-MNIST Benchmark \(Image Classification\) | Papers With Code](#)

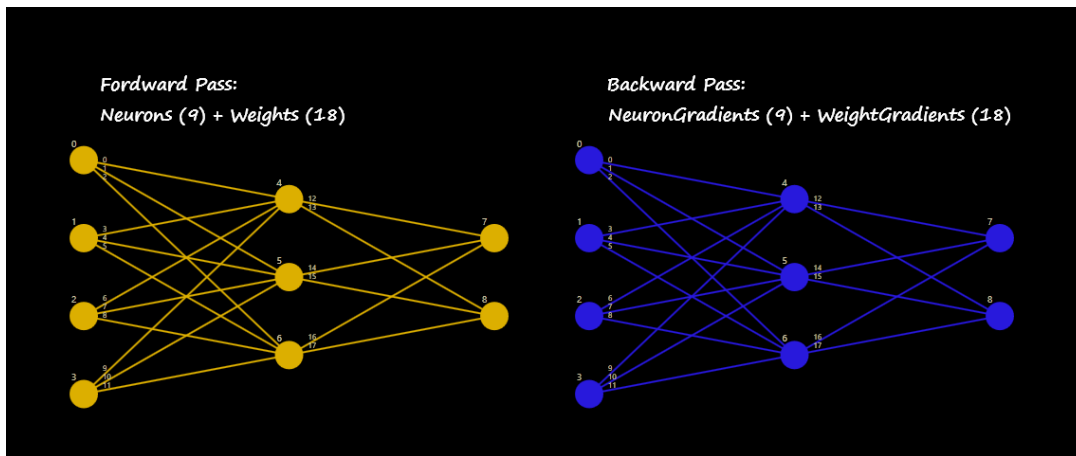
With 92.45% accuracy on position 12. That's almost in the top ten. It feels pretty good for not so much effort.

Further steps could be to optimize the CNN architecture in some ways. For example, I like to group the output maps instead of connecting them to the entire NN input layer.

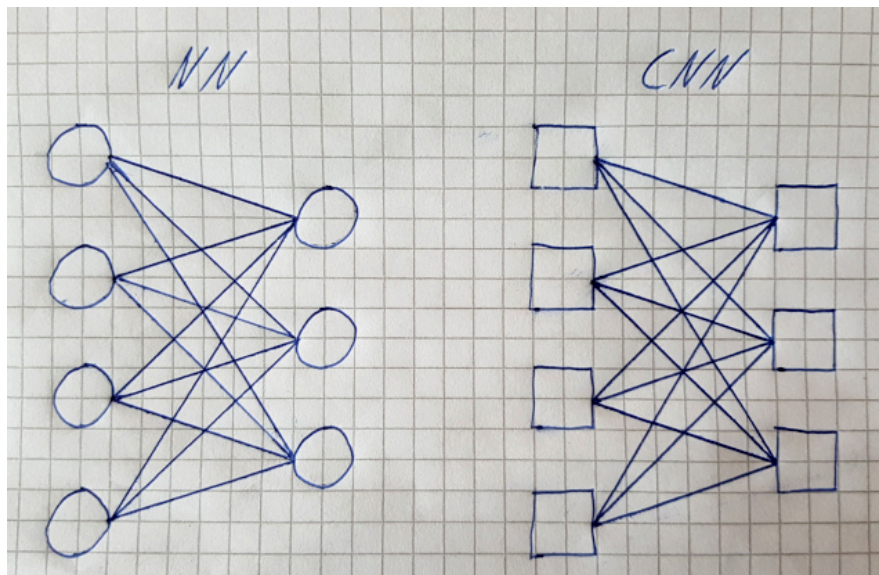
Fully connecting $3,388 * 400$ neurons requires a huge amount of 1,355,200 weights. But to connect 28 maps consisting of 121 neurons to 20 output neurons as a group, resulting in a stack of output neurons of $20 * 28 = 560$, then the cost goes down to $28 * 121 * 20 = 67,760$ weights and works great.

AlexNet or other architectures can also help to get ideas on how to improve the prediction model. The paper "A ConvNet for the 2020s" summarizes many nice ideas. The next step was ConvNeXt V2, so it goes on.

Bonus Insights



Understanding the logic and how to run this structure on the machine is crucial. By this simple understanding, I would name the delta of the kernel `kernelGradient`.



A classic convolutional neural network runs in the same way through the structure as the fully connected neural network. Most important are the costs. Input neuron times weight, summed on an output neuron, can be seen as a simple cost of 1. The CNN replaces the neuron by maps of many neurons. The weight is replaced by a kernel consisting of many weights. The cost of this operation can be more than 10,000! For example a filter = 3, stride = 1, input map = $28 * 28$, output map = $26 * 26 = 676$. So one output map with 676 neurons created by its 9 kernel weights would cost 6,084 operations in total. InputMap dimension 28 times filter dimension 6, as used in on the first CNN layer: $6 * 6 * 23 * 23$ should result in 19,044 operations. Right?

Derivation of Backpropagation in Convolutional Neural Network (CNN)

Zhifei Zhang

University of Tennessee, Knoxville, TN

October 18, 2016

Abstract— Derivation of backpropagation in convolutional neural network (CNN) is conducted based on an example with two convolutional layers. The step-by-step derivation is helpful for beginners. First, the feedforward procedure is claimed, and then the backpropagation is derived based on the example.

1 Feedforward

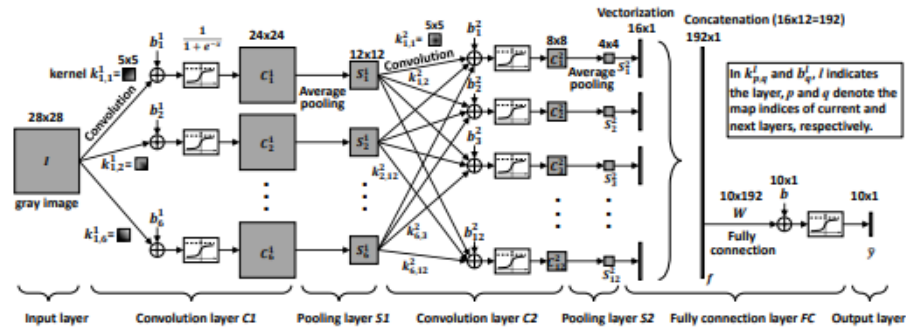
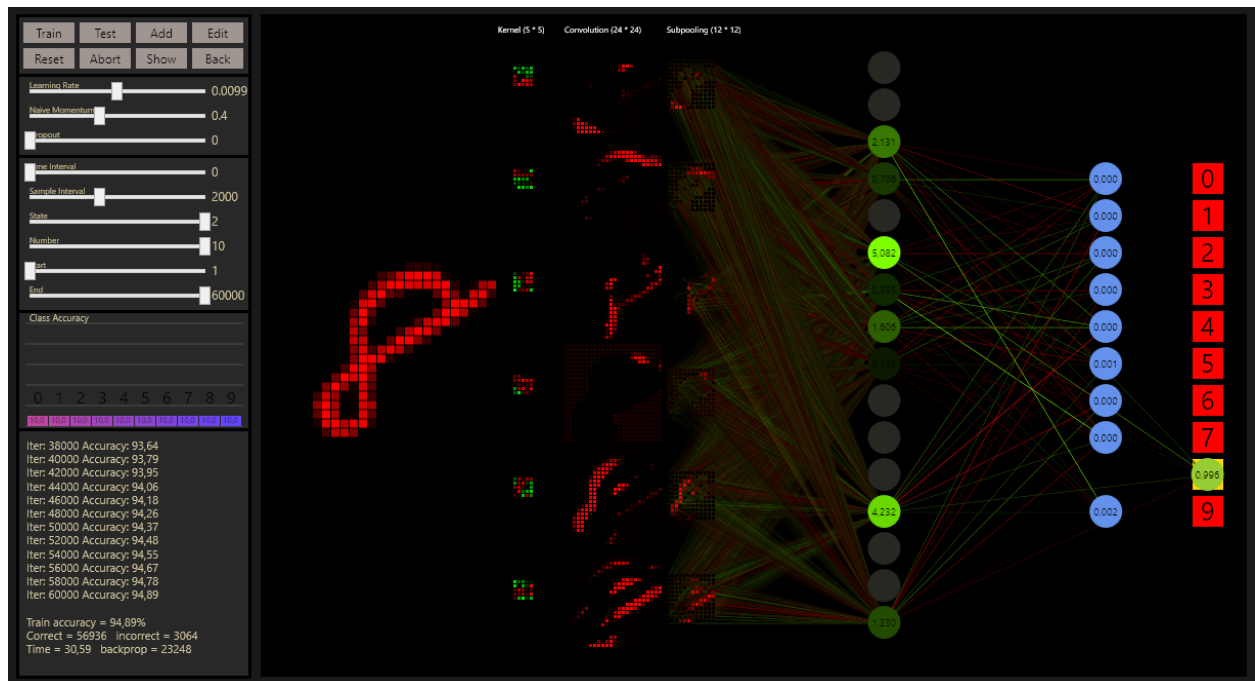


Figure 1: The structure of CNN example that will be discussed in this paper. It is exactly the same to the structure used in the demo of Matlab DeepLearnToolbox [1]. All later derivation will use the same notations in this figure.

[Derivation of Backpropagation in Convolutional Neural Network \(CNN\)](#)

This was the first serious paper I found to grasp the idea of convolutional neural networks with backpropagation. There is a lot of math, but much of it is repetitive. Be careful when implementing the logic from math to code to keep the arrays in bounds.

Step 9.0: CNN Visualization

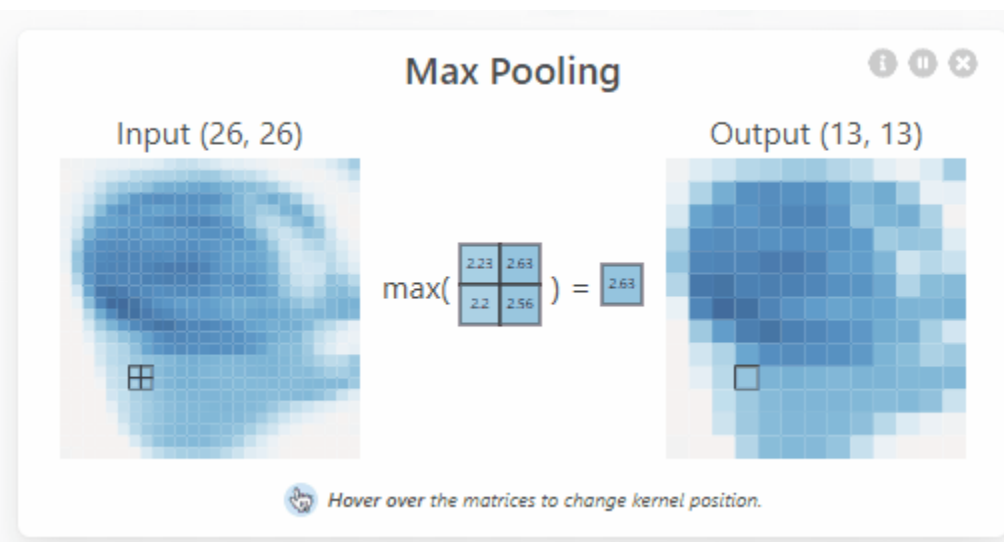
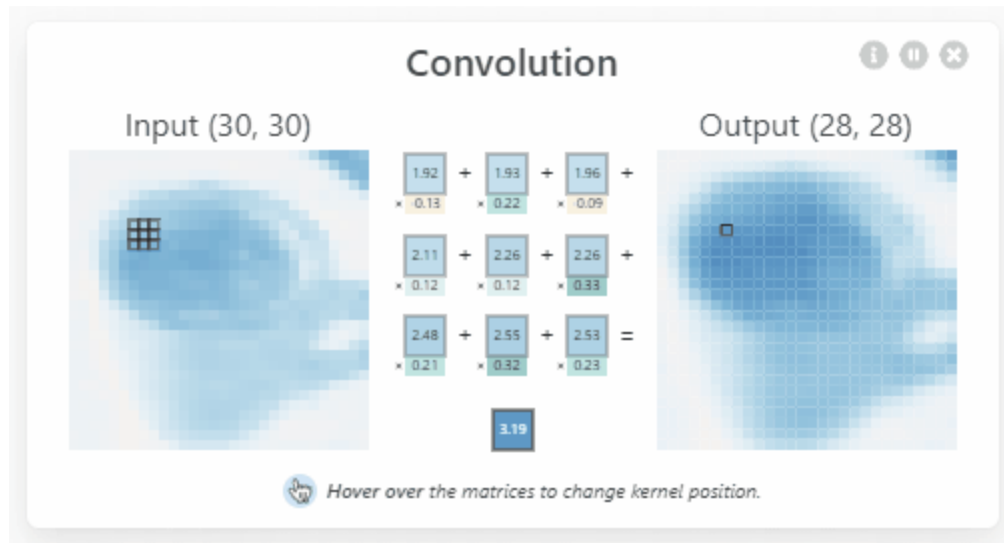


This was one of my first simplified visualizations of the paper. It was really interesting to see how the CNN implementation works. The kernels are already trained. Where each kernel seems to take a different part of the input image to create its convolutional feature map. It looks like a deconstruction of the original image.

CNN Visualization with 2 Layers



Two CNN layers as used in the demo code are harder to visualize, but it works in the same way. It's a really raw realization. But this visual help provides a nice way to optimize the signal flow through the network. There are no guidelines to do that, but seeing the effect of different implementation details can help to build a better performing model. I hope I did this well.



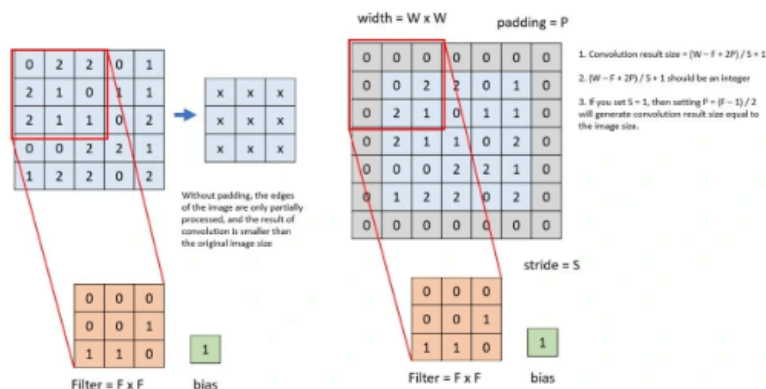
Credits to: <https://github.com/poloclub/cnn-explainer>

Convolution Image Size, Filter Size, Padding and Stride

Posted on May 30, 2018 by jamesdmccaffrey

A convolutional neural network (CNN) applies a filter to an image in a very tricky way. When you use a CNN you have to be aware of the relationship between the image size, the filter size, the size of the padding around the image, and the distance the filter moves (the stride) during convolution.

Without image padding, the pixels on the edge of the image are only partially processed (which may be OK), and the result of convolution will be smaller than the original image size (usually not good). I'll cut to the chase and give the key facts.



Suppose an image has size $W \times W$, the filter has size $F \times F$, the padding is P , and the stride is S . Then:

1. The result size of a convolution will be $(W - F + 2P) / S + 1$. For example, if an image is 100×100 , a filter is 6×6 , the padding is 7, and the stride is 4, the result of convolution will be $(100 - 6 + (2)(7)) / 4 + 1 = 28 \times 28$.
2. Therefore, the quantity $(W - F + 2P) / S + 1$ should be an integer, and so $(W - F + 2P)$ should be evenly divisible by S . This will never be a problem if $S = 1$ but could be a problem if S is greater than 1.
3. If you set $S = 1$ (very common), then by setting $P = (F - 1) / 2$ the result size of convolution will be the same as the image size (which is usually what you want). If S is greater than 1, then you need to adjust P and/or F if you want to retain the original image size.

When I sat down to write this blog post it was my intention to explain exactly where these relation equations come from. But I quickly realized that providing a full explanation would take a couple pages of text (at least). So, instead I'll just say that when you work with convolution, you can't just use any values for your filter, padding, and stride.

[Convolution Image Size, Filter Size, Padding and Stride | James D. McCaffrey](#)

[Convolutional Neural Networks for MNIST Data Using PyTorch in Visual Studio Magazine | James D. McCaffrey](#)

[A PyTorch Convolution Layer Worked Example | James D. McCaffrey](#)

[Yet Another PyTorch ResNet Example | James D. McCaffrey](#)

[The Five Neural Network Weight Initialization Algorithms | James D. McCaffrey](#)

[The Math Derivation of the Softmax with Max and Log Tricks | James D. McCaffrey](#)

[The Max Trick when Computing Softmax | James D. McCaffrey](#)

https://github.com/grensen/how_to_build

<https://github.com/grensen/multi-core>

https://github.com/grensen/good_vs_bad_code

https://github.com/grensen/how_to_train

[High-Performance Neural Network Benchmark in Parallel with SIMD and Modern C# Optimization on MNIST](#)

[My first released CNN, quite reduced it reaches 99.20% accuracy on MNIST.](#)

[A short demo that shows how to create, train, test, save and reload a CNN model, which ranks under the top 50 models on MNIST test.](#)

[▶ Inside a Neural Network - Computerphile](#)

[▶ CNN: Convolutional Neural Networks Explained - Computerphile](#)

[▶ A friendly introduction to Convolutional Neural Networks and Image Recognition](#)

[▶ But what is a convolution?](#)

[▶ Convolutions | Why X+Y in probability is a beautiful mess](#)

[▶ But what is a neural network? | Chapter 1, Deep learning](#)

[▶ How convolutional neural networks work, in depth](#)

[▶ Yann LeCun: Deep Learning, ConvNets, and Self-Supervised Learning | Lex Fridman Podcast #36](#)

Ilya Sutskever (Tel Aviv University's Blavatnik School of Computer Science, minute 4):

"Progress in AI is a game of faith. The more faith you have, the more progress you can make.

So, if you have a very, very large amount of faith, you can make the most progress. And it sounds like I'm joking, but I'm not. You have to believe. You have to believe in the idea and push on it. The more you believe, the harder you can push, and that's what leads to progress. Now, it's important that the thing you believe in is correct, but with that caveat, it's all about the belief."