Digit recognition Demo

Intro

Recognizing a handwritten digit is intuitive for the human eye, as we can effortlessly discern between a scribbled '7' and a rushed '2'. Yet, for machines, this is an intricate challenge. Imagine trying to teach a toddler to identify numbers, but with matrices of pixels, algorithms and various model architectures. In this demo, our journey begins with straightforward linear models. As we progress, we explore feed-forward neural networks, which use interconnected web of nodes to resemble how our brains process new information. Then, we navigate through convolutional neural networks (CNNs), designed to mimic the human visual system. With these tools in mind, we hope you can have a deeper understanding of neural networks and appreciate the power of machine learning algorithms.

For this demo, we will be using the MNIST dataset, which contains a vast collection of handwritten digits that have been used in the machine learning community for years. Each image in this dataset is 28×28 pixels in size and represents a grayscale image of a single digit. So why should we care about classifying these images? Imagine a post office sorting through zip codes or a bank reading handwritten checks - having a machine to automatically recognize the digits with high accuracy can be very useful. The dataset contains $60,000$ training samples and $10,000$ testing samples, with a similar number of samples for each digit.

Load data and Pre-Process

We first load the required packages and the data:

```python
import numpy as np
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Here, we load the MNIST dataset, which is split between training and testing split. More specifically, `x_train` is a `(60000 x 28 x 28)` numpy array of integers [0, 255], and `y_train` is a `(60000, )` numpy array of labels [0, 10). To simplify our task, we consider binary classification where we zoom into two digits: 4 and 9. This choice allows us to understand the fundamental concepts of classification before tackling all ten digits. Let's define some global variables:

```python
DIG_A, DIG_B = 4, 9
SIDE = 28
MAX_PIX_VAL = 255.
```

and keep only the images of digits 4 and 9. To make our computations more stable and faster, we normalize our image data to have values between 0 and 1.

```
indices = np.logical_or((y_train == DIG_A), (y_train == DIG_B))
x_train = x_train[indices]
y_train = y_train[indices]
x_train = x_train / MAX_PIX_VAL
N = len(x_train)
```

```
Sanity check:
assert len(x_train) == len(y_train)
assert x_trian.shape == (N, )
assert set(y_train) == {DIG_A, DIG_B}
```

It is crucial to randomize our dataset to ensure that the model does not accidentally learn any pattern from the order in which samples are presented. We also fix the seed to ensure code reproducibility.

```
np.random.seed(42)
indices = np.arange(len(x_train))
np.random.shuffle(indices)
x_train = x_train[indices]
y_train = y_train[indices]
```

Let us define the function `close_enough` to check if the data is appropriately normalized. Furthermore, given that there are roughly equal numbers of each digit in the dataset, we should have close to 12,000 samples.

```
close_enough = lambda a, b: abs(b-a) < 1e-6
```

```
Sanity check:
assert close_enough(np.min(x_train), 0.)
assert close_enough(np.max(x_train), 1.)
assert abs(N - 12000) < 500
```

Evaluate Predictors

Next, we define some functions to evaluate how good our model is using accuracy and cross-entropy loss. The `judge` function is a handy tool that lets us get both metrics for any predictors we pass in:

```
def accuracy(predicted_labels, true_ys):
    return np.mean([1. if l==y else 0. for l, y in zip(predicted_labels, true_ys)])

def cross_entropy_loss(predicted_probs, true_ys):
    return np.mean([-np.log(p if y==DIG_B else 1.-p) for p, y in zip(predicted_probs, true_ys)])

def judge(predictor, xs, ys):
    probs = [predictor(x) for x in xs]
    labels = [DIG_B if p > .5 else DIG_A for p in probs]
    acc = accuracy(labels, ys)
    loss = cross_entropy_loss(probs, ys)
    return {'acc': acc, 'loss': loss}
```

Dummy Predictors

Before diving into complicated predictive models, let us define some dummy predictor functions that completely ignore the input image. They can either be very certain or uncertain about their prediction, or a fifty-fifty coin toss. These models, while naive in their predictions, give us a perspective on what's achievable if we took a very basic approach to predicting our digits.

```
very_sure_A = lambda x: .01
very_sure_B = lambda x: .99
maybe_its_A = lambda x: .4
maybe_its_B = lambda x: .6
fifty_fifty = lambda x: .5
```

`very_sure_A` and `very_sure_B` are predictors that are, as their names suggest, very sure about their predictions. The former always thinks the digit is 'A' (or in our case, '4'), while the latter is adamant it's 'B' (or '9'). These models don't even look at the input image; they just confidently guess one digit. We can assert that the combined accuracy of the two models must sum up to one, because if one is correct, the other has to be incorrect, or vice versa.

```
vsa = judge(very_sure_A, x_train, y_train)['acc']
vsb = judge(very_sure_B, x_train, y_train)['acc']
assert close_enough(vsa + vsb, 1.)
```

Next, we test these dummy predictors against a single instance. Using `very_sure_A` against individual instances of 'A' and 'B', it should confidently get the former right and the latter wrong.

```
vsa = judge(very_sure_A, x_train[:1], [DIG_A])['acc']
vsb = judge(very_sure_A, x_train[:1], [DIG_B])['acc']
assert close_enough(vsa, 1.)
assert close_enough(vsb, 0.)
```

Lastly, we examine the loss of these predictors, which gives us a measure of how off the predictors are from the truth.

```python
vsa = judge(very_sure_A, x_train, y_train)['loss']
vsb = judge(very_sure_B, x_train, y_train)['loss']
mia = judge(maybe_its_A, x_train, y_train)['loss']
mib = judge(maybe_its_B, x_train, y_train)['loss']
ffl = judge(fifty_fifty, x_train, y_train)['loss']
```

Cross-entropy loss quantifies the difference between two probability distributions, typically the true distribution and our model's predictions. For the binary classification task, cross-entropy loss heavily penalizes a confident but incorrect prediction. Recall the formula for cross-entropy loss for binary classification:

$$L = -y \cdot \log(p) - (1 - y) \cdot \log(1 - p)$$

where y is the true label, and p is the predicted probability of the label being 1. Therefore, for `very_sure_A`, the predictor encounters a near-zero loss for correct predictions, but a heavy penalty when making a confident but incorrect prediction. On the other hand, `maybe_its_A` makes predictions with lower confidence, thus encounters a more moderate loss for both correct and incorrect predictions.

`fifty_fifty` always suggests that either digit is equally probable, thus the cross-entropy loss is simply $-\log(1/2) = \log 2$. Given that the dataset contains similar number of samples for each label, `fifty_fifty` yields predictions that are closest to the baseline distribution. As the predictors become more extreme, each incorrect prediction yields heavier penalty that outweights a lower loss of correction prediction.

```python
assert ffl < mia < vsa
assert ffl < mib < vsb
assert close_enough(ffl, np.log(2))
```

Q: what if the distribution of the two samples is skewed? For example, how will the cross-entropy loss relationship change if 90% of the samples are 'A'?

Linear Models

The journey of creating a linear model often starts with the activation function. In this case, we're using the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

In order to ensure numerical stability, we clip the value of z between -15 and 15 as follows:

```python
clip = lambda z: np.clip(z, -15, 15)
sigmoid = lambda z: 1./(1.+np.exp(-clip(z)))
```

With our activation function defined, let's build our predictor! The linear prediction is simply the weighted sum of the input data passed through the sigmoid function. Here, we flatten the input into a 1D array, given that it is a 28×28 array.

```
def linear_predict(w, x):
    return sigmoid(w.dot(x.flatten()))
```

We initialize the w with random weights, similar to how a neural network is initialized. This is the linear mapping from the input. Here, we initialize w with values drawn from a normal distribution normalized by diving the square root of the total number of pixels. This is to prevent any random weight from starting too large.

```
w = np.random.randn(SIDE * SIDE) / np.sqrt(SIDE * SIDE)
```

Let's perform a sanity check on our linear implementation so far. Consider the predictor with our random weights w, we obtain a prediction based on the weighted sum of the values in x. If we feed the same sample, the weighted sum becomes the exact negative of the previous weighted sum because the weights are the negatives of the original weights. This will essentially mirror the input value of the sigmoid function from one side to the other. Since the sigmoid function is symmetric around the vertical line $z = 0$ (where $\sigma(z) = 0.5$), if we get a value greater than 0.5 using w, we must get a value less than 0.5 using the negative weights -w, and vice versa. In other words, flipping the sign of w guarantees that the two predictors will always make opposite predictions for any sample, and their respective confidence scores (the raw outputs of the sigmoid) will sum to 1.

```
vsa = judge(lambda x: linear_predict(+w, x), x_train, y_train)['acc']
vsb = judge(lambda x: linear_predict(-w, x), x_train, y_train)['acc']
assert close_enough(vsa+vsb, 1.)
```

We can also consider a special case of zero weights. In this case, all predictions have a 50-50 chance, and the associated loss must be approximately $\log 2$.

```
ffl = judge(lambda x: linear_predict(0*w, x), x_train, y_train)['loss']
assert close_enough(ffl, np.log(2))
```

Let's do another interesting experiment. Suppose we create a random image x using the random weights. In this case, the dot product between w and x will always be positive, thus the prediction will always be in favor of digit B. To verify this:

```
x = w.reshape(SIDE, SIDE)
vsa = judge(lambda x: linear_predict(w, x), [x], [DIG_A])['acc']
vsb = judge(lambda x: linear_predict(w, x), [x], [DIG_B])['acc']
```

```
assert close_enough(vsa, 0)
assert close_enough(vsb, 1)
```

Linear Backpropagation

The term "backpropagation" is typically used for neural networks, but we can apply the same concept to linear models. In essence, it is the math behind adjusting the weights and learning from errors to get a better model. Consider the loss function:

$$L = \mathrm{CEL}(\sigma(w \cdot x))$$

where CEL denotes the cross-entropy loss and σ denotes the sigmoid function. Here, we use chain rule to obtain the gradient of the loss function with respect to the weights. Let $z \triangleq w \cdot x$, and $p \triangleq \sigma(z)$, we have:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Recall the cross-entropy loss for binary classifications, we have:

$$L(p, y) = \begin{cases} -\log p & \text{if } y = \text{DIG\_B} \\ -\log(1-p) & \text{otherwise} \end{cases}$$

Taking the first derivative, we have:

$$\frac{\partial L}{\partial p} = \begin{cases} -1/p & \text{if } y = \text{DIG\_B} \\ 1/(1-p) & \text{otherwise} \end{cases}$$

Next, we consider the derivative of sigmoid (computation details omitted).

$$\frac{\partial p}{\partial z} = p(1-p)$$

Finally, the derivative of z with respect to w is simply x (flattened). Combining everything above, we have:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial w} = \begin{cases} (p-1) \cdot x & \text{if } y = \text{DIG\_B} \\ p \cdot x & \text{otherwise} \end{cases}$$

Therefore, we can define the linear backpropogration function as follows:

```python
def linear_backprop(w, x, y):
    z = w.dot(x.flatten())
    p = sigmoid(z)

    dl_dp = -(1 if y == DIG_B else -1) / (p if y == DIG_B else 1-p)
    dp_dz = p * (1-p)
    dz_dw = x.flatten()

    dl_dw = dl_dp * dp_dz * dz_dw
    return dl_dw
```

Let's do a sanity check to make sure that the linear predictor is actually learning something. Using just one sample, we should expect the loss to decrease for each epoch. We also define a more generalized version of the function `close_enough` to compare two arrays.

```
close_enough = lambda a, b: np.linalg.norm(np.array(b-a).flatten()) < 1e-6

for _ in range(10):
    w = np.random.randn(SIDE*SIDE) / np.sqrt(SIDE*SIDE)
    x = x_train[0]
    y = y_train[0]

    g = linear_backprop(w, x, y)

    before = judge(lambda xx: linear_predict(w, xx), [x], [y])['loss']
    w = w - 0.01 * g
    after = judge(lambda xx: linear_predict(w, xx), [x], [y])['loss']
    assert after < before
```

Building the SGD engine

Next, we define a function that allows us to retrieve the next training example using a global index variable. We also initialize the random weights, define some learning parameters and number of epochs.

```
idx = 0

def next_training_example():
    global idx
    xy = x_train[idx], y_train[idx]
    idx += 1
    idx %= N
    return xy

w = np.random.randn(SIDE * SIDE) / np.sqrt(SIDE * SIDE)
LEARNING_RATE = 0.1
epochs = 10000
```

Now that everything is working, we can finally build our first SGD engine!

```
for t in range(epochs):
    x, y = next_training_example()
    g = linear_backprop(w, x, y)
    w = w - LEARNING_RATE * g

    if t % 1000:
        continue

    ms = judge(lambda x: linear_predict(w, x), x_train, y_train)
    print('at step {:6d}'.format(t) +
          'tr acc {:4.2f} tr loss {:5.3f}'.format(ms['acc'], ms['loss']))
```

We train the linear predictor with a fixed learning rate for 10,000 epochs. For every 1,000 epoch, we evalute the model to obtain its training accuracy and loss. We get the following output (note that your results may differ due to randomness):

```
at step      0tr acc 0.51 tr loss 0.959
at step   1000tr acc 0.95 tr loss 0.143
...
at step   9000tr acc 0.97 tr loss 0.115
at step  10000tr acc 0.96 tr loss 0.137
```

As shown above, the model achieves a 96% accuracy! However, the model seems to reach its optimal performance in only 1,000 epochs. The loss indicates that the model is unstable, as it fluctuates between 0.11 and 0.17. This may be due to a relatively large learning rate, which prevents the training from stabilizing. Therefore, we can try a learning rate scheduler that decays the learning rate slowly overtime.

```
LR = LEARNING_RATE * 1000. / (1000. + t)
```

Additionally, increase the number of epochs, and get the following output (again, your results may differ):

```
at step      0tr acc 0.50 tr loss 1.127
at step  10000tr acc 0.97 tr loss 0.095
...
at step  90000tr acc 0.97 tr loss 0.088
at step 100000tr acc 0.97 tr loss 0.087
```

As shown above, we reached a lower loss at 0.087! From our exploration, it's evident that even a linear model, despite its inherent simplicity, can exhibit profound capabilities in addressing complicated tasks like image classification, achieving a 97% accuracy.

Vanilla Models

Let's move on to vanilla feed-forward neural networks! There are many high-level deep learning frameworks (ie. `PyTorch` and `TensorFlow`) that can simplify the following code in just a few lines, but building the network from scratch enables us to grasp the fundamentals. We first build the following network architecture:

```
x
h0 --------> z1 ---> h1 --------> z2 ---> h2 --------> z3 ------> p
 |
 |
 |           |       |
 |           |       |            |       |
 |     C*    | lrelu |     B*     | lrelu |     A*     | sigmoid |
 |           |       |            |       |
 |           |       |            1
 |           1
 |
 1
D0          D1      D1          D2      D2           D3        1
```

The input x does through the network, layer by layer, to the output. Recall that a feed-forward neural network consists of one input layer, a fixed number of hidden layers, and one output layer. In our example, the network consists of two hidden layers, each equipped with leaky ReLU as its activation function. Leaky ReLU is a modified version of ReLU which addresses the issue of vanishing gradients.

$$\mathrm{lrelu}(z) = \max\left(z/10, z\right)$$

The last row of the diagram indicates the dimension of layer at each layer. Since our input is a 28×28 image, the input to the network is a flattened array of 784 dimensions. For simplicity, we set $\mathrm{D1} = \mathrm{D2} = 32$, which is more so an arbitrary choice. In practice, the number of neurons in hidden layers is often chosen to be a power of 2, which can result in computational efficiency due to how hardware architectures like GPUs are designed. Notice the "1" below each layer? That is the bias term added to each layer. Now, we build this vanilla network and initialize it with random weights:

```python
D0 = SIDE * SIDE
D1, D2, D3 = 32, 32, 1

def vanilla_init():
    A = np.random.randn(    D2) / np.sqrt( 1 + D2)
    B = np.random.randn(D2, D1) / np.sqrt(D2 + D1)
    C = np.random.randn(D1, D0) / np.sqrt(D1 + D0)
    return (A, B, C)
```

The weights are initialized using the Xavier initialization, which scales the weights by the inverse of the square root of the sum of the input and output sizes. This helps to achieve a variance of activations that remain the same across layers, facilitating better convergence during training. Next, we define the activation function and the feed-forward prediction:

```
lrelu = lambda z: np.maximum(z/10, z)

def vanilla_predict(abc, x):
    A, B, C = abc

    h0 = x.flatten()
    z1 = C.dot(h0)
    h1 = lrelu(z1)

    z2 = B.dot(h1)
    h2 = lrelu(z2)

    z3 = A.dot(h2)
    p = sigmoid(z3)
    return p
```

This function passes the input x through the network, layer by layer, with the sigmoid function at the end to ensure that the output is between 0 and 1, indicating the prediction probability. Similarly, we perform a sanity check on a dummy predictor:

```
A, B, C = vanilla_init()
vsa = judge(lambda x: vanilla_predict((+A, B, C), x), x_train, y_train)['acc']
vsb = judge(lambda x: vanilla_predict((-A, B, C), x), x_train, y_train)['acc']
assert close_enough(vsa+vsb, 1.)
```

Here, `vsa` computes the accuracy using the initial random weights, whereas `vsb` computes the accuracy when the sign of the weights of the output layer is flipped, essentially leading to opposite predictions for the same input. Therefore, the accuracy of the two predictors must sum up to one. Next, consider another predictor where `A` is set to zero. In this case, the output layer's activation are forced to be near 0.5, turning the model into a fifty-fifty predictor. Recall that the loss for this type of predictor should have loss of $\log 2$.

```
ffl = judge(lambda x: vanilla_predict((0*A, B, C), x), x_train, y_train)['loss']
assert close_enough(ffl, np.log(2))
```

To further understand the effects of these weights, we can perform an interesting experiment. Suppose we force all the weights to be positive, then the values in each neuron would be positive, leading to a prediction that always favors `DIG_B`. We can verify this as follows:

```
x = x_train[0]
y = y_train[0]
A = np.abs(A)
B = np.abs(B)
C = np.abs(C)

acc_ppp = judge(lambda x: vanilla_predict((A,  B,  C), x), [x], [DIG_B])['acc']
assert close_enough(acc_ppp, 1.)
```

Suppose we flip one of the three matrices, such that all entries in the matrix are negative. Therefore, the inputs and outputs of that particular layer would be opposite, leading to a prediction that always favors `DIG_A`. We can verify this as follows:

```
acc_ppn = judge(lambda x: vanilla_predict((A,  B, -C), x), [x], [DIG_B])['acc']
acc_pnp = judge(lambda x: vanilla_predict((A, -B,  C), x), [x], [DIG_B])['acc']
assert close_enough(acc_ppn, 0.)
assert close_enough(acc_pnp, 0.)
```

Similarly, if we flip two of the matrices, the two negative weights will end up negating each other's effect!

```
acc_pnn = judge(lambda x: vanilla_predict((A, -B, -C), x), [x], [DIG_B])['acc']
assert close_enough(acc_pnn, 1.)
```

Vanilla Backpropagation

Now, we are ready to implement the back-propagation for the vanilla model. Recall that leaky ReLU is defined as:

$$\mathrm{lrelu}(z) = \max\left(z/10, z\right) = \begin{cases} z & \text{if } z \geq 0 \\ z/10 & \text{otherwise} \end{cases}$$

We compute its gradient:

$$\frac{\partial \mathrm{lrelu}}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ 0.1 & \text{otherwise} \end{cases}$$

Observe that leaky ReLU is not differentiable at $z = 0$, although it is typically defined to be the same as the negative side to avoid having undefined values. In this example, we define it to be the average of the two derivatives (ie. 0.55). Note that in practice, the gradient at $z = 0$ does not really matter due to floating-point arithmetic in Python. The probability of getting an exact value of 0 is extremely small. Therefore, we can define the derivative as:

```python
step = lambda z: np.heaviside(z, 0.5)
dlrelu_dz = lambda z: 0.1 + (1. - 0.1) * step(z)
```

Note that:

$$\texttt{np.heaviside(z, c)} = \begin{cases} 0 & \text{if } z < 0 \\ c & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Consider the weights `abc = (A, B, C)`, and given sample x with label y. For each update, the function first performs a forward pass through the network to compute the probability of class `DIG_B` of the given network for `x`.

```python
def vanilla_backprop(abc, x, y):
    A, B, C = abc

    h0 = x.flatten()
    z1 = C.dot(h0)
    h1 = lrelu(z1)

    z2 = B.dot(h1)
    h2 = lrelu(z2)

    z3 = A.dot(h2)
    p = sigmoid(z3)
```

Next, the function computes the gradients of the loss with respect to the parameters by applying the chain rule backward through the network, from the output layer to the input layer. This part calculates how much each weight contributed to the error in the output and thus, how much the weights should be adjusted. Here, `np.outer` computes the gradient for each weight in the layer based on how much changing that specific weight would affect the loss. The resulting matrix of the outer product has the same shape as the weight matrix, and each element of the result represents the partial derivative of the loss with respect to the corresponding weight in the layer.

```python
    dl_dz3 = p - (1 if y == DIG_B else 0)
    dl_dh2 = dl_dz3 * A
    dl_dz2 = dl_dh2 * dlrelu_dz(z2)
    dl_dh1 = dl_dz2.dot(B)
    dl_dz1 = dl_dh1 * dlrelu_dz(z1)

    dl_dA = dl_dz3 * h2
    dl_dB = np.outer(dl_dz2, h1)
    dl_dC = np.outer(dl_dz1, h0)

    return dl_dA, dl_dB, dl_dC
```

Next, we define a displacement function that updates the parameters of the network. For each parameter, the corresponding gradient is multiplied by the learning rate then added to the current parameter value, performing a gradient descent update.

```
def vanilla_displace(abc, coef, g):
    A, B, C = abc
    gA, gB, gC = g
    return (A + coef * gA,
            B + coef * gB,
            C + coef * gC)
```

Combining everything above, let's train the vanilla network!

```
abc = vanilla_init()

for t in range(T*15+1):
    x, y = next_training_example()
    g = vanilla_backprop(abc, x, y)
    LR = LEARNING_RATE * 4000. / (4000. + t)
    abc = vanilla_displace(abc, -LR, g)

    if t % 1000:
        continue

    ms = judge(lambda x: vanilla_predict(abc, x), x_train, y_train)
    print('at step {:6d}'.format(t) +
          'tr acc {:4.2f} tr loss {:5.3f}'.format(ms['acc'], ms['loss']))
```

We get the following output:

```
at step      0tr acc 0.64 tr loss 0.678
at step   1000tr acc 0.95 tr loss 0.145
...
at step  14000tr acc 0.99 tr loss 0.044
at step  15000tr acc 0.98 tr loss 0.052
```

This indicates that the network reaches a training loss of around 0.05, with an $98 - 99\%$ accuracy! Additionally, we can implement the idea of *momentum*, which also considers gradients from previous updates. We first initialize the momentum to zero after we initialize the weights of the model (at line 2): `m = vanilla_displace(abc, -1., abc)`. This is a hacky way to initialize momentum with the same shape as model parameters. Next, we change the model updates to:

```
1 beta = 0.9
2 m = vanilla_displace(m, beta - 1, m)
3 m = vanilla_displace(m, 1, g)
4 abc = vanilla_displace(abc, -LR, m)
```

The parameter `beta` is the momentum coefficient, which controls how much of the previous gradients to be considered. For example, when $\beta = 0.9$, it means 90% of the previous velocity is combined with 10% of the current gradient to update the parameters. Line 2 indicates that we are "forgetting" 10% of the previous velocity (thus 90% remaining), then we the current gradient to the velocity. Then we update the parameters based on the velocity. In practice, we typically set the momentum parameter to some value between 0.8 and 0.99. Using this technique, we can sometimes result in faster convergence and higher accuracy (although the effects are not as profound in this example).

```
at step      0tr acc 0.51 tr loss 0.690
at step   1000tr acc 0.95 tr loss 0.147
...
at step  14000tr acc 0.98 tr loss 0.056
at step  15000tr acc 0.99 tr loss 0.043
```

Building a CNN

For the last model, we will be building a convolutional neural network (CNN). A CNN is a specialized type of neural network designed for processing grid-structured input data such as images, where spatial hierarchies and local patterns are crucial. Unlike fully-connected neural networks where each neuron in one layer is connected to every neuron in the next layer, CNNs use convolutional layers to preserve the spatial relationship between pixels by learning image features using small squares of input data. We define the architecture as below:

```
                   height x width x channels

x                     28 x 28 x 1
       avgpool                                         2 x 2
h0                    14 x 14 x 1
       conv                                weight C    5 x 5 x 8 x 1    stride 2 x 2
```

```
z1                    5 x 5 x 8
      lrelu
h1                    5 x 5 x 8
      conv                                weight B   1 x 1 x 4 x 8   stride 1 x 1
z2                    5 x 5 x 4
      lrelu
h2                    5 x 5 x 4
      dense                               weight A   1 x (5 x 5 x 4)
z3                            1
      sigmoid
p                             1
```

Let's break down the architecture layer by layer, from top to bottom:

1. input `x`: the input is an image with dimensions `28 x 28` with 1 channel, since it is a grayscale image.
2. avgpool: this layer performs average pooling with a `2 x 2` window, effectively reducing the spatial dimensions by a factor of 2. After pooling, the dimension becomes `14 x 14 x 1`.
3. conv: the first convolution layer has filters of size `5 x 5`, 8 output channels with a `2 x 2` stride. Given the input size of `14 x 14`, the feature map size becomes:
$$\frac{14-5}{2}+1=5$$
   The output has dimension `5 x 5 x 8`. Leaky ReLU does not change the dimensions.
4. The second convolution layer has filters of size `1 x 1`, 4 output channels and operating on 8 input channels with a stride of `1 x 1`. The spatial dimensions remain the same, but the number of channels changes to 4. The output has dimension `5 x 5 x 4`. Leaky ReLU does not change the dimensions.
5. We use a fully-connected layer that reduces the dimensions to a single value, followed by a sigmoid activation function which squashes the output between 0 and 1, which is what we need for binary classification tasks.

Now, we define the functions for average pooling and convolution layers.

```python
def avgpool2x2(x):
    H, W, C = x.shape
    return ( x[0:H:2, 0:W:2]
            +x[0:H:2, 1:W:2]
            +x[1:H:2, 0:W:2]
            +x[1:H:2, 1:W:2]) / 4

def conv(x, weights, stride=1):
    H, W, C = x.shape
    KH, KW, OD, ID = weights.shape
    assert C == ID
    HH, WW = int((H - KH + 1) / stride), int((W - KW + 1) / stride)
    return np.array(
        [[
            np.tensordot(
                weights,
                x[h:h+KH, w:w+KW],
                ((0, 1, 3), (0, 1, 2))
            )
            for w in range(0, WW*stride, stride)]
        for h in range(0, HH*stride, stride)]
    )
```

The function avgpool2x2 performs 2x2 pooling, which means it averages the values of 2x2 adjacent pixels and produces a pooled output, hence reducing the spatial dimensions by a factor of 2. The function conv performs convolution operation between the input x and weights with a specified stride. The assertion test ensures that the number of channels in the input is the same as the number of channels to the input depth of the weights ID. The returned value includes the dot products between the weights and the respective regions of the input tensor, considering the stride. It iterates over the height and width of the possible positions of the filter on the input tensor and calculates the corresponding output for each position, eventually creating an array representing the output feature map. Now, we perform some sanity checks:

```
aa = np.ones((8, 12, 7))
pp = np.ones((4, 6, 7))
assert close_enough(avgpool2x2(aa), pp)
ww = np.ones((3, 3, 5, 7))
cc = (3*3*7)*np.ones((6, 10, 5))
assert close_enough(conv(aa, ww, stride=1), cc)
```

We first initialize 'aa' as an array of all ones with shape 8 x 12 x 7 (height x width x channel). When we perform a 2x2 average pooling on it, we effectively shrink the dimensions by a factor of 2. Since all values are ones, the average value in each pooling region is also one. Next, 'ww' represents the weights of the convolutional layer and has dimensions 3 x 3 x 5 x 7 (kernel height x kernel width x output channels x input channels). We perform a convolution operation on 'aa' with weights 'ww' and stride of 1. The output dimensions can be calculated as $(8-3+1, 12-3+1) = (6, 10)$, thus matching the dimensions of 'cc'. Since the weights 'ww' are all ones, each value in the output 'cc' should be the sum of 3 x 3 x 7 ones from the input 'aa', hence each value in 'cc' should be $3 \cdot 3 \cdot 7 = 63$. These two sanity checks test the scaling and shapes of the functions above. Now, we test if the results match in terms of their orientations:

```
bb = np.array([1 * np.eye(4), 3 * np.eye(4)])
pp = np.array([[[1, 1, 0, 0], [0, 0, 1, 1]]])
assert close_enough(avgpool2x2(bb), pp)
ww = np.zeros([2, 2, 1, 4])
ww[0, 0, :, :] = 1 + np.arange(4)
cc = np.array([1, 2, 3])[np.newaxis, :, np.newaxis]
assert close_enough(conv(bb, ww, stride=1), cc)
```

'`bb`' is equivalent to two `4 x 4` identity matrices stacked together, with the second scaled by 3. When `avgpool2x2` is applied to '`bb`', it performs average pooling over `2 x 2` non-overlapping windows. Each channel is an input of dimension `2 x 4`, in which the i'th column of the i'th channel being 1 and 3, e.g.

$$C_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{bmatrix}$$

After average pooling with `2 x 2` windows, we have:

$$C_1 = C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}, C_3 = C_4 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Stacking the channels together, we yield an array with dimensions `1 x 2 x 4`:

$$pp = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

For the second sanity check, '`w`' represents the weights of the convolution layer with dimension `2 x 2 x 1 x 4` (kernel height x kernel width x output channels x input channels). Observe that only the top-left values of the weights are non-zero. Therefore, the convolution for each channel would be the sum of the diagonal of '`bb`' scaled by the corresponding weight in '`ww[0, 0]`', which results in the array '`cc`'.
Now, we compute the gradients of the convolution layer. We write two helper functions that, when given the outputs to the convolution layer, give us the gradients with respect to the weights or the inputs to the layer.

```
def Dw_conv(x, weights_shape, dl_dconv, stride=1):
    H, W, C = x.shape
    KH, KW, OD, ID = weights_shape
    assert C == ID
    HH, WW = int((H-KH+1)/stride), int((W-KW+1)/stride)
    assert dl_dconv.shape == (HH, WW, OD)

    HS, WS = HH*stride, WW*stride
    dl_dw = np.array(
        [[
            np.tensordot(
                dl_dconv,
                x[dh:dh+HS:stride, dw:dw+WS:stride],
                ((0, 1), (0, 1))
            )
            for dw in range(KW)]
        for dh in range(KH)]
    )
    return dl_dw
```

This function gives us the derivative of the loss with respect to the weights of the convolution layer. Each entry of the derivative:

$$\frac{\partial L}{\partial W_{ij}} = \text{np.tensordot}(\frac{\partial L}{\partial C_{ij}}, x_{\text{sliced}}, ((0,1),(0,1)))$$

computes the sum of element-wise products of the gradients `dl_dconv` and the corresponding input values `x_sliced`.

```
def Dx_conv(x_shape, weights, dl_dconv, stride):
    H, W, C = x_shape
    KH, KW, OD, ID = weights.shape
    assert C == ID
    HH, WW = int((H-KH+1)/stride), int((W-KW+1)/stride)
    assert dl_dconv.shape == (HH, WW, OD)

    dl_dx = np.zeros((H, W, ID), dtype=np.float32)
    for h in range(KH):
        for w in range(KW):
            dl_dx[h:h+HH*stride:stride, w:w+WW*stride:stride] += (
                  np.tensordot(
                      dl_dconv,
                      weights[h, w],
                      ((2, ), (0, ))
                  )
            )
    return dl_dx
```

The second function gives us the derivative of the loss with respect to the inputs to the convolution layer. Each entry:

$$\frac{\partial L}{\partial x_{ij}} = \sum_{h,w} \frac{\partial L}{\partial C_{i-h,j-w}} \cdot W_{hw}$$

is the sum over all positions in the kernel, and accumulates the contribution to the gradient at each input location (i,j) due to each weight in the kernel. With these building blocks, let's build the full back-propagation for the network:

```
 1 def conv_backprop(abc, x, y):
 2     A, B, C = abc
 3 
 4     h0 = avgpool2x2(x[:, :, np.newaxis])
 5 
 6     z1 = conv(h0, C, stride=2)
 7     h1 = lrelu(z1)
 8 
 9     z2 = conv(h1, B, stride=1)
10     h2 = lrelu(z2)
11 
12     z3 = A.dot(h2.flatten())
13     p = sigmoid(z3)
14 
15     dl_dz3 = p - (1 if y == DIG_B else 0)
16     dl_dh2 = dl_dz3 * A.reshape(h2.shape)
17     dl_dz2 = dl_dh2 * dlrelu_dz(z2)
18     dl_dh1 = Dx_conv(h1.shape, B, dl_dz2, stride=1)
19     dl_dz1 = dl_dh1 * dlrelu_dz(z1)
20 
21     dl_dA = dl_dz3 * h2.flatten()
22     dl_dB = Dw_conv(h1, B.shape, dl_dz2, stride=1)
23     dl_dC = Dw_conv(h0, C.shape, dl_dz1, stride=2)
24 
25     return (dl_dA, dl_dB, dl_dC)
```

Next, we define the prediction function and the displacement function. We omit the implementations here because the prediction function simply returns the `p` from above (line 13), and the displacement function is identical to the one implemented in the vanilla models. We also initialize the random weights (similarly, using normalized Gaussian distributions):

```
def conv_init():
    A = np.random.randn(5 * 5 * 4) / np.sqrt(1+5*5*4)
    B = np.random.randn(1, 1, 4, 8) / np.sqrt(4+1*1*8)
    C = np.random.randn(5, 5, 8, 1) / np.sqrt(8+5*5*1)
    return (A, B, C)
```

Finally, let's train the convolutional neural network!

```
abc = conv_init()
m = conv_displace(abc, -1., abc)

```

```
for t in range(10001):
    x, y = next_training_example()
    g = conv_backprop(abc, x, y)
    LR = LEARNING_RATE * 4000. / (4000. + t)

    abc = conv_displace(abc, -LR, g)

    if t % 1000:
        continue

    ms = judge(lambda x: conv_predict(abc, x), x_train, y_train)
    print('at step {:6d}'.format(t) +
          'tr acc {:4.2f} tr loss {:5.3f}'.format(ms['acc'], ms['loss']))
```

We get the following performance:

```
at step      0tr acc 0.53 tr loss 0.692
at step   1000tr acc 0.94 tr loss 0.170
...
at step   9000tr acc 0.98 tr loss 0.049
at step  10000tr acc 0.99 tr loss 0.044
```

Comparing to previous models, we are able to reach an accuracy of 99% slightly faster!

Evaluation on test set

In the final section, we will assess the performance of our three models using the testing dataset. The objective is to validate that our models are not merely overfitting to the training dataset but are also capable of generalizing effectively to unseen data. To this end, we load both the training and the testing datasets. Mirroring our previous preprocessing approach, we retain only the two digits of interest, shuffle the samples, and normalize the images.

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

def preprocess(x, y):
    indices = np.logical_or((y == DIG_A), (y == DIG_B))
    x = x[indices]
    y = y[indices]
    x = x / MAX_PIX_VAL

    indices = np.arange(len(x))
    np.random.shuffle(indices)
    x = x[indices]
    y = y[indices]

    return x, y

np.random.seed(42)
x_train, y_train = preprocess(x_train, y_train)
x_test, y_test = preprocess(x_test, y_test)
```

Next, we will evaluate each model's performance on the test set. A consistent set of parameters are used across all experiments, in which we use gradient descent with momentum to minimize loss.

```
T = 15001
LEARNING_RATE = 0.01
ANNEAL_T = 4000.
BETA = 0.9
```

We also use the following interface for each model to call the corresponding functions.

```
functions = {
    'linear': [linear_init, linear_backprop, linear_displace, linear_predict],
    'vanilla': [vanilla_init, vanilla_backprop, vanilla_displace, vanilla_predict],
    'conv': [conv_init, conv_backprop, conv_displace, conv_predict]
}
```

We compare the models as follow:

```
for model in functions.keys():
    init, backprop, displace, predict = functions[model]

    w = init()
    m = displace(w, -1., w)

    for t in range(T):
        x, y = next_training_example()
        g = backprop(w, x, y)
        LR = LEARNING_RATE * ANNEAL_T / (ANNEAL_T + t)

        m = displace(m, BETA - 1, m)
        m = displace(m, 1, g)
        w = displace(w, -LR, m)

        if t % 1000:
            continue

        train = judge(lambda x: predict(w, x), x_train, y_train)
        test = judge(lambda x: predict(w, x), x_test, y_test)

        print('at step {:6d} '.format(t) +
              'tr acc {:4.3f} loss {:5.3f} '.format(train['acc'], train['loss']) +
              'te acc {:4.3f} loss {:5.3f}'.format(test['acc'], test['loss']))
```

We obtain the following loss and accuracies at the end of the training:

Model	Training Acc	Training Loss	Testing Acc	Testing Loss
Linear	0.970	0.092	0.969	0.095
Vanilla	0.978	0.060	0.976	0.069
CNN	0.987	0.039	0.988	0.035

As shown above, the CNN model yields the best performance, both on the training and the testing dataset. In practice, we focus more on the testing accuracy and loss as they represent the performance on unseen observations.
Now, let's perform a "stress" test, in which we test the performance of each model under extreme circumstances. Here, we provide only $1,000$ training samples with added Gaussian noise:

```
x = x[:1000]
x = x + np.random.randn(*x.shape)
x = np.clip(x, 0, 1)
```

We obtain the following loss and accuracies at the end of the training:

Model	Training Acc	Training Loss	Testing Acc	Testing Loss
Linear	1.000	0.012	0.895	0.655
Vanilla	0.926	0.286	0.893	0.647
CNN	0.821	0.394	0.914	0.225

Here, we observe some interesting results. The linear model is able to predict all samples in the training dataset correctly, achieving an 100% accuracy rate. The training accuracy decreases as the complexity of the model increases. However, the linear model performs worse on the testing dataset, with the CNN achieving about 2% higher accuracy rate. The linear model and the vanilla model has similar performance on the testing dataset. Note that you may get different results as you run this comparison multiple times, but if given enough time to fully converge (without overfitting), the statements above generally hold true.

Conclusion

If you're reading this, that means...congratulations! You have just taught a machine to classify hand-written digits! Not only that, the models are able to achieve near- or super-human performance, as human accuracy is reported to be around 97-98% accuracy. In this exercise, we built linear models, vanilla feed-forward networks, and convolution neural networks completely from scratch without relying on external libraries. This enables us to understand some important concepts and the math behind neural networks, such as forward and backward propagation, gradient descent and weight updates.

You may be surprised by the impressive performance of a simple linear model paired with a single leaky ReLU. This highlights the power of non-linearity, which can reveal underlying complicated structures. Some state-of-the-art deep neural networks have shown over 99.7% accuracy on classifying all 10 digits. It is worth noting that while machine learning models can achieve higher accuracy than humans on this specific task, it doesn't necessarily mean they "understand" the data in the same way humans do, as they can be easily fooled with adversarial examples. While MNIST is remains highly popular, researchers typically use more complex datasets, such as ImageNet or CIFAR-10, for standard benchmarking.

We hope you enjoyed this demo and found it helpful. While our focus here is on binary classification, keep in mind that the MNIST dataset includes samples for ten distinct digits, offering many opportunities for extension. We encourage you to experiment with different model architectures, hyperparameters, and even delve into multi-class classification to further enhance the model's performance. Happy modeling!