```
from tkinter import *

class ImmutableEntry(Entry):

    def __init__(self, parent, text):
        Entry.__init__(self, parent, disabledforeground = "black",
                       justify = "center")
        self.insert(0, text)
        self.config(state = "disabled")

from Tensor import Tensor
from Tensor import product
import random
import pickle
import os

class InputLayer:

    layer_type = "input"

    def __init__(self, tensor, reshape=False, activation=Tensor.dummy):
        self.reshape = reshape
        self.activation = activation
        self.forward_pass(tensor)

    def forward_pass(self, tensor):
        if self.reshape:
            tensor = tensor.reshape(tensor.dims[:1]+self.reshape)
```

```
        self.tensor = tensor
        self.result_dims = tensor.dims
        return tensor


class ConvolutionalLayer:

    layer_type = "convolutional"

    def __init__(self, kernel_size, kernel_count, prev_layer, stride=1,
                 activation=Tensor.leaky_relu, pretrained=False):
        self.dims = [kernel_count, kernel_size, kernel_size]
        self.pretrained = pretrained
        if pretrained:
            self.tensor = Tensor(array=pretrained)
        else:
            self.tensor = Tensor(dims=self.dims)
        self.activation = activation
        self.result_dims = Tensor.convolve_dims(prev_layer.result_dims,
                                                self.tensor.dims, stride)

    def forward_pass(self, tensor_in, train=True):
        tensor_out, mask = tensor_in.convolve(self.tensor)
        tensor_out = self.activation(tensor_out)
        self.result_dims = tensor_out.dims
        self.mask = mask
        return tensor_out
```

```
    def backward_pass(self, delta_in, hidden, learning_rate=1, momentum=0):
        if self.pretrained:
            return None, None
        delta_in = delta_in.reshape(self.result_dims)
        t = hidden.reverse_convolve(delta_in, fs=self.tensor.dims[-1])
        self.tensor += t*learning_rate


class FullyConnectedLayer:

    layer_type = "fully_connected"

    def __init__(self, width, prev_layer, activation=Tensor.tanh,
                 array=False, drop_out=0):
        self.dims = [product(prev_layer.result_dims[1:]), width]
        self.drop_out_percentage = drop_out
        if array:
            self.tensor = Tensor(array=array)
        else:
            self.tensor = Tensor(dims=self.dims)
        self.result_dims = [prev_layer.result_dims[0], width]
        self.prev_layer = prev_layer
        self.activation = activation

    def forward_pass(self, tensor_in, train=True):
        if len(tensor_in.dims)<3:
```

```
            resh = tensor_in
        else:
            new_dims = [tensor_in.dims[0], product(tensor_in.dims[1:])]
            resh = tensor_in.reshape(new_dims)
        if train:
            dims = [tensor_in.dims[0], self.dims[-1]]
            mask = Tensor.generate_zero_mask(dims, self.drop_out_percentage)
            return self.activation(resh.dot(self.tensor))*mask
        else:
            return self.activation(resh.dot(self.tensor))

    def backward_pass(self, delta_in, hidden, learning_rate, momentum=0):
        hidden = hidden.reshape([hidden.dims[0], product(hidden.dims[1:])])
        transpose = self.tensor.transpose()
        p = (100-self.drop_out_percentage)/100
        delta = delta_in.dot(transpose)*self.activation(hidden, deriv=True)
        update = (hidden.transpose().dot(delta_in))*learning_rate*p
        momentum_update = update+momentum
        self.tensor += momentum_update
        self.tensor = self.tensor.max_norm_constrain()
        return delta, update


class OutputLayer(FullyConnectedLayer):

    layer_type = "output_layer"
```

```
    def __init__(self, outputs, prev_layer,
                 activation=Tensor.tanh, array=False):
        super().__init__(outputs, prev_layer, activation, array)

    def forward_pass(self, tensor_in, train=False):
        return super().forward_pass(tensor_in, train=False).soft_max()

    def backward_pass(self, delta_in, hidden, learning_rate, momentum=0):
        return super().backward_pass(delta_in, hidden, learning_rate, momentum)


class MaxPoolLayer:

    layer_type = "max_pool"

    def __init__(self, prev_layer, window_size=2, stride=2):
        self.result_dims = prev_layer.result_dims[:-2]
        self.result_dims.append(prev_layer.result_dims[-2]//stride)
        self.result_dims.append(prev_layer.result_dims[-1]//stride)
        self.window_size = window_size
        self.stride = stride

    def forward_pass(self, tensor_in, train=False):
        result, self.mask = tensor_in.max_pool(self.window_size, self.stride)
        self.result_dims = result.dims
        return result
```

```
    def backward_pass(self, delta_in, hidden, learning_rate=1, momentum=0):
        delta_in = delta_in.reshape(self.result_dims)
        delta = delta_in.reverse_max_pool(self.mask, self.window_size, self.stride)
        return delta, None


class Network:

    def __init__(self, momentum_decay=0.9):
        self.layers = []
        self.high_score = 0
        self.test_cycles = 0
        self.learning_rate = 1
        self.momentum_decay = momentum_decay

    def finalise(self):
        self.updates = []
        for layer in self.layers[1:][::-1]:
            t = layer.layer_type != "max_pool"
            i = Tensor.generate_zero_mask(layer.tensor.dims, 100)if t else None
            self.updates.append(i)

    def add_layer(self, layer):
        self.layers.append(layer)

    def convolutional_layers(self):
        result = []
```

```
    for i, l in enumerate(self.layers):
        if l.layer_type=="convolutional":
            result.append((i, l.tensor))
    return result

def max_layers(self):
    result = []
    for i, l in enumerate(self.layers):
        if l.layer_type=="max_pool":
            result.append(i)
    return result

def forward_pass(self, tensor=False, train=True):
    pass_results = [self.layers[0].forward_pass(tensor)]
    for i, layer in enumerate(self.layers[1:]):
        pass_results.append(layer.forward_pass(pass_results[i],
                                               train=train))
    self.pass_results = pass_results

def backward_pass(self, tensor, batch):
    error = tensor-self.pass_results[-1]
    delta = error*self.pass_results[-1]
    zipped = zip(self.layers[::-1], self.pass_results[:-1][::-1])
    for i, (layer, hidden) in enumerate(zipped):
        delta, update = layer.backward_pass(delta, hidden,
                                            self.learning_rate/batch,
                                            self.updates[i])
```

```
        if update is not None:
            self.updates[i]=self.updates[i]*self.momentum_decay+update*0.6
        else:
            self.updates[i] = 0

def batch(self, inputs, outputs, batch_size, same=False):
    inputs = inputs.array if isinstance(inputs, Tensor) else inputs
    outputs = inputs.array if isinstance(inputs, Tensor) else outputs
    in_array, out_array = [], []
    for i in range(batch_size):
        r = i if same else random.randrange(len(inputs))
        in_array.append(inputs[r])
        out_array.append(outputs[r])
    return (Tensor(array=in_array), Tensor(array=out_array))

def train(self, inputs, outputs, learning_rate=0.1, cycles=25, batch=25):
    self.learning_rate = learning_rate
    for i in range(cycles):
        print(i, end=" ")
        tensor_in, tensor_out = self.batch(inputs, outputs, batch)
        self.forward_pass(tensor_in)
        self.backward_pass(tensor_out, batch)
    self.test_cycles += 1

def test(self, inputs, outputs, batch=100):
    tensor_in, tensor_out = self.batch(inputs, outputs, batch, True)
    predicted = self.predict(tensor_in)
```

```
        score = 0
        for a, b in zip(predicted.array, tensor_out.array):
            score += 1 if a==b else 0
        return score/batch*100

    def predict(self, tensor):
        self.forward_pass(tensor, train=False)
        return self.pass_results[-1].one_hot()

    def save(self, score=0):
        if score>=self.high_score:
            try:
                os.remove("Model{0}.Pickle".format(self.high_score))
            except:
                pass
            pickle.dump((self.layers, score),
                        open("Model{0}.Pickle".format(score), "wb"))
            self.high_score = score

    def load(self, score=0):
        self.layers, self.high_score = pickle.load(open("Model{0}.pickle"
                                                        .format(score), "rb"))
```

```
from tkinter import *
from tkinter import font

class Popup(Toplevel):

    def __init__(self, parent, relx, rely, relwidth, relheight,title="Warning", alf=False):
        Toplevel.__init__(self, parent)
        self.parent = parent
        self.title(title)
        self.grid_columnconfigure(0, weight = 1)
        self.set_size(parent, relx, rely, relwidth, relheight)
        self.grab_set()

    def grab_set(self):
```

```
        while True:
            try:
                Toplevel.grab_set(self)
                break
            except:
                pass

    def set_size(self, parent, relx, rely, relwidth, relheight):
        g = parent.geometry()
        a = g.split("x")
        b = a[1].split("+")
        pw, ph, px, py = int(a[0]), int(b[0]), int(b[1]), int(b[2])
        w, h = int(pw*relwidth), int(ph*relheight),
        x, y = int(px+pw*relx), int(py+ph*rely)
        self.geometry("{0}x{1}+{2}+{3}".format(w, h, x, y))

    def display_message(self, messages):
        for row, message in enumerate(messages):
            entry = Entry(self, disabledforeground = "black", justify ="center")
            entry.insert(0, message)
            entry.config(state = "disabled")
            entry.grid(row = row, column = 0, sticky = "nsew")

import random
import numbers
import math
import pickle
import copy
```

```
def product(array):
    if isinstance(array, Tensor):
        array = array.array
    result = 1
    for i in array:
        result *= i
    return result

class Tensor():

    """
    dims in order for 2D tensor/matrix:rows/height,columns/width
    dims in order for 3D tensor:layers/depth,rows/height,columns/width
    for an ND tensor follow this pattern for declaration of more spacial
    dimensions.
    """
    def __init__(self, dims=False, array=False):
        if isinstance(array, list):
            self.array = array
            self.dims = Tensor.recursive_find_dims(self.array, [], True)
        elif isinstance(dims, list) and isinstance(dims[0], int):
            self.dims = dims
            self.array = Tensor.recursive_build(self.dims)
        else:
            raise AttributeError("""dimensions or an array must be
                                 provided to initialisation""")
```

```
def max_norm_constrain(array, max_value=4, first=True):
    r = []
    if first:
        if isinstance(array, Tensor):
            array = array.array
    if isinstance(array, list):
        for a in array:
            r.append(Tensor.max_norm_constrain(a, max_value, False))
    else:
        r = max(-4, min(4, array))
    if first:
        return Tensor(array=r)
    return r


def generate_zero_mask(dims, percent=30, first=True):
    r = []
    if len(dims)>1:
        for _ in range(dims[0]):
            r.append(Tensor.generate_zero_mask(dims[1:], percent, False))
    else:
        for _ in range(dims[0]):
            r.append(1 if random.randrange(0, 100)>=percent else 0)
    if first:
        return Tensor(array=r)
    return r
```

```
def weight_statistics(array, lower=-4, upper=4, step=1, r=False):
    first = not r
    if first:
        r = {}
        if isinstance(array, Tensor):
            array = array.array
        for i in range(lower, upper+1, step):
            r[i] = 0
    if isinstance(array, list):
        for a in array:
            Tensor.weight_statistics(a, lower, upper, step, r)
    else:
        r[min(r, key = lambda x: abs(x-array))] += 1
    if first:
        return r

def recursive_find_dims(array, dims, first=False):
    dims.append(len(array))
    if isinstance(array[0], list):
        Tensor.recursive_find_dims(array[0], dims)
    if first:
        return dims

def recursive_build(dims):
    array = []
    if len(dims)>1:
        for _ in range(dims[0]):
```

```
            array.append(Tensor.recursive_build(dims[1:]))
        else:
            for _ in range(dims[0]):
                array.append(random.uniform(-1, 1))
        return array

    def recursive_str(array):
        string = ""
        for i in array:
            if isinstance(array[0], list):
                string += Tensor.recursive_str(i)+"\n"
            else:
                t = str(i)
                if len(t)<3:
                    t += " "*(3-len(t))
                elif len(t)>3:
                    t = t
                string += "{0} ".format(t)
        return string

    def __str__(self):
        return Tensor.recursive_str(self.array)

    def __repr__(self):
        return "Tensor of dimensions: {0}".format(self.dims)

    def recursive_math(array, other, func, start=False):
```

```
    if isinstance(array, Tensor):
        array.check_dims(other)
        array = array.array
    other = other if not isinstance(other, Tensor) else other.array
    to_return = []
    if isinstance(array[0], list):
        if isinstance(other, list):
            for i, j in zip(array, other):
                to_return.append(Tensor.recursive_math(i, j, func))
        else:
            for i in array:
                to_return.append(Tensor.recursive_math(i, other, func))
    else:
        if isinstance(other, list):
            for i, j in zip(array, other):
                to_return.append(func(i, j))
        else:
            for i in array:
                to_return.append(func(i, other))
    if start:
        return Tensor(array=to_return)
    return to_return

def __add__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x+y, True)

def __radd__(self, other):
```

```
    return Tensor.__add__(self, other)

def __sub__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x-y, True)

def __rsub__(self, other):
    return Tensor.__sub__(self, other)

def __mul__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x*y, True)

def __rmul__(self, other):
    return Tensor.__mul__(self, other)

def __div__(self, other):
    return Tensor.recursive_math(self, other, lambda x,y:x/y, True)

def __rdiv__(self, other):
    return Tensor.__mul__(self, other)

def recursive_relu(array, deriv=False, leaky=0):
    r = []
    if isinstance(array[0], list):
        for a in array:
            r.append(Tensor.recursive_relu(a))
    elif deriv:
        for a in array:
```

```
            r.append(1 if a>0 else leaky)
        else:
            for a in array:
                r.append(a if a>0 else leaky*a)
        return r

    def relu(self, deriv=False):
        return Tensor(array=Tensor.recursive_relu(self.array, deriv))

    def leaky_relu(self, deriv=False, leaky=-0.1):
        return Tensor(array=Tensor.recursive_relu(self.array, deriv, leaky))

    def check_dims(self, other):
        t = "Dimension Error: Tensors of dimensions {0} and {1} don't match"
        if isinstance(other, Tensor):
            if other.dims != self.dims:
                raise Exception(t.format(self.dims, other.dims))

    def sum(array):
        result = 0
        if isinstance(array[0], list):
            for i in array:
                result += Tensor.sum(i)
        else:
            for i in array:
                result += i
        return result
```

```
def recursive_dot():
    pass


def dot(self, other):
    if not isinstance(other, Tensor):
        raise TypeError("Argument to dot product must be a Tensor object.")
    if len(other.dims)!=2:
        raise ValueError("Dot only implemented for 2D Tensor argument.")
    if self.dims[-1] != other.dims[-2]:
        t = "{0}\nself: {1} other: {2}"
        t1 = "Rows and Columns of matrices do not align"
        t = t.format(t1, self.dims, other.dims)
        raise ValueError(t)
    x = []
    for a in range(self.dims[-2]):
        y = []
        for b in range(other.dims[-1]):
            z = 0
            for c in range(self.dims[-1]):
                z += self.array[a][c]*other.array[c][b]
            y.append(z)
        x.append(y)
    return Tensor(array=x)


def recursive_polarise(array):
    if isinstance(array, list):
```

```
        if isinstance(array[0], list):
            a = []
            for i in array:
                a.append(Tensor.recursive_polarise(i))
            return a
        else:
            a = []
            for i in array:
                a.append(-1 if i<=0 else 1)
            return a

def rec_dummy(array):
    r = []
    if isinstance(array, list):
        for a in array:
            r.append(Tensor.rec_dummy(a))
    else:
        return array
    return r

def dummy(self, *args, **kwargs):
    return Tensor(array=Tensor.rec_dummy(self.array))

def polarise(self):
    array = Tensor.recursive_polarise(self.array)
    self.array = array
```

```
def recursive_flatten(self, array):
    for i in array:
        if isinstance(array[0], list):
            self.recursive_flatten(i)
        else:
            self.flattened.append(i)

def flatten(self):
    self.flattened = []
    self.recursive_flatten(self.array)
    return Tensor(array=self.flattened)

def recursive_max_pool(array, window_size=2, stride=2):
    a = []
    if not isinstance(array[0][0], list):
        for i in range(len(array)//stride):
            b = []
            i *= stride
            for j in range(len(array[0])//stride):
                j *= stride
                v = -99999999999999
                for k in range(window_size):
                    for l1 in range(window_size):
                        try:
                            if i+k>=0 and j+l1>=0:
                                v = max(v, array[i+k][j+l1])
                        except IndexError:
```

```
                    pass
                b.append(v)
            a.append(b)
    else:
        for ar in array:
            a.append(Tensor.recursive_max_pool(ar, window_size, stride))
    return a


def make_max_pool_mask(self, array, window_size, stride, first=False):
    a = []
    if not isinstance(array[0][0], list):
        a = []
        for _ in array[0]:
            b = []
            for _ in array:
                b.append(0)
            a.append(b)
        for i in range(len(array)//stride):
            i *= stride
            for j in range(len(array[0])//stride):
                j *= stride
                v = -99999999999999
                for k in range(window_size):
                    for l1 in range(window_size):
                        try:
                            if i+k>=0 and j+l1>=0:
                                v = max(v, array[i+k][j+l1])
```

```
                except IndexError:
                    pass
            for k in range(window_size):
                for l1 in range(window_size):
                    try:
                        if i+k>=0 and j+l1>=0:
                            if array[i+k][j+l1] >= v and v>0:
                                a[i+k][j+l1] = 1
                    except IndexError:
                        pass
    else:
        for ar in array:
            a.append(self.make_max_pool_mask(ar, window_size, stride))
    if first:
        return Tensor(array=a).reshape(self.dims)
    return a

def max_pool(self, window_size=2, stride=2):
    mask = self.make_max_pool_mask(self.array, window_size, stride, True)
    pooled = Tensor(array=Tensor.recursive_max_pool(self.array,
                                        window_size, stride))
    return (pooled, mask)

def recursive_reverse_max_pool(array, mask, window_size,
                        stride, first=False):
    a = []
    if not isinstance(array[0][0], list):
```

```
        for i in range(len(mask)):
            b = []
            for j in range(len(mask[0])):
                b.append(array[i//stride][j//stride]*mask[i][j])
            a.append(b)
    else:
        for ar, ms in zip(array, mask):
            a.append(Tensor.recursive_reverse_max_pool(ar, ms, window_size,
                                                       stride))
    if first:
        return Tensor(array=a)
    return a


def reverse_max_pool(self, mask, window_size=2, stride=2):
    return Tensor.recursive_reverse_max_pool(self.array, mask.array,
                                             window_size, stride, True)

def recursive_reshape(self, out_array, counter, in_array, dims):
    if counter+1==len(dims):
        for i in range(dims[counter]):
            out_array.append(in_array[self.counter])
            self.counter += 1
    else:
        for i in range(dims[counter]):
            out_array.append(self.recursive_reshape([], counter+1,
                                                    in_array, dims))
```

```
    return out_array

def reshape(self, dims):
    self.counter = 0
    if product(self.dims)!=product(dims):
        raise ValueError("Tensor will not fit into specified dimensions.")
    in_array = self.flatten().array
    return Tensor(array=self.recursive_reshape([], 0, in_array, dims))

def convolve_dims(image, kernel, stride):
    return image[:-2]+[kernel[0], image[-2]//stride, image[-1]//stride]

def rec_reverse_con(a, a0, fs, stride, pl, pr, pt, pb):
    u = []
    if isinstance(a[0][0], list) and isinstance(a0[0][0], list):
        for t, t0 in zip(a, a0):
            u.append(Tensor.rec_reverse_con(t, t0, fs, stride, pl,
                                            pr, pt, pb))
    elif isinstance(a[0][0], list):
        for t in a:
            u.append(Tensor.rec_reverse_con(t, a0, fs, stride,
                                            pl, pr, pt, pb))
    elif isinstance(a0[0][0], list):
        for t in a0:
            u.append(Tensor.rec_reverse_con(a, t, fs, stride,
                                            pl, pr, pt, pb))
    else:
```

```
        for _ in range(fs):
            u0 = []
            for _ in range(fs):
                u0.append(0)
            u.append(u0)
        for i in range(-pt, len(a[0])-pb):
            for j in range(-pl, len(a[0])-pr):
                for l in range(fs):
                    for m in range(fs):
                        try:
                            if i+l>0 and j+m>0:
                                u[l][m] += a[i+l][j+m]*a0[i][j]
                        except IndexError:
                            pass
    return u

def merge(self):
    x = Tensor(array=self.array[0])
    for i in self.array[1:]:
        x += Tensor(array=i)
    x = x.__div__(len(self.array))
    return x

def rec_zero_pad(array, l, r, t, b):
    x = []
    if isinstance(array[0][0], list):
        for a in array:
```

```
            x.append(Tensor.rec_zero_pad(a, l, r, t, b))
    elif isinstance(array[0], list):
        for i in range(t):
            y = []
            for j in range(len(array[0])+l+r):
                y.append(0)
            x.append(y)
        for a in array:
            y = []
            for i in range(l):
                y.append(0)
            for i in a:
                y.append(i)
            for i in range(r):
                y.append(0)
            x.append(y)
        for i in range(b):
            y = []
            for j in range(len(array[0])+l+r):
                y.append(0)
            x.append(y)
    return x


def zero_pad(self, l, r, t, b):
    return Tensor(array=Tensor.rec_zero_pad(self.array, l, r, t, b))
```

```
    def reverse_convolve(self, other, fs=3, l=False, r=False,
                         t=False, b=False, stride=1, pad=False):
        if not isinstance(other, Tensor):
            raise TypeError("""Argument to convolution operation
                               must be a Tensor object.""")
        if pad:
            l, r, t, b = pad, pad, pad, pad
        elif not l:
            l, r, t, b = 1, 1, 1, 1
        asd =  Tensor(array=Tensor.rec_reverse_con(self.array, other.array, fs, stride, l, r,
t, b))
        asd = asd.merge()
        return asd



    def rec_con(a1, a2, stride, pl, pr, pt, pb):
        r = []
        m = []
        if isinstance(a1[0][0], list) and isinstance(a2[0][0], list):
            for t1 in a1:
                r1 = []
                m0 = []
                for t2 in a2:
                    af, mf = Tensor.rec_con(t1, t2, stride, pl, pr, pt, pb)
                    r1.append(af)
                    m0.append(mf)
                r.append(r1)
```

```
            m.append(m0)
        elif isinstance(a1[0][0], list):
            for t1 in a1:
                af, mf = Tensor.rec_con(t1, a2, stride, pl, pr, pt, pb)
                r.append(af)
                m.append(mf)
        elif isinstance(a2[0][0], list):
            for t2 in a2:
                af, mf = Tensor.rec_con(a1, t2, stride, pl, pr, pt, pb)
                r.append(af)
                m.append(mf)
        else:
            output_rows = len(a1)//stride-len(a2[0])+1
            output_columns = len(a1[0])//stride-len(a2)+1
            for i in range(-pt, output_rows+pb):
                i *= stride
                e = []
                m0 = []
                for j in range(-pl, output_columns+pr):
                    j *= stride
                    x = 0
                    m1 = []
                    for a in range(len(a2)):
                        m2 = []
                        for b in range(len(a2[0])):
                            try:
                                if a+i>=0 and b+j>=0:
```

```
                        v = a1[a+i][b+j]*a2[a][b]
                        m2.append(v)
                        x += v
                    else:
                        m2.append(0)
                except IndexError:
                    m2.append(0)
                    pass
            m1.append(m2)
        m0.append([[y/x if x else 0 for y in z] for z in m1])
        e.append(x)
    m.append(m0)
    r.append(e)
return r, m

def convolve(self, other, l=False, r=False, t=False, b=False,
             stride=1, pad=False):
    if not isinstance(other, Tensor):
        raise TypeError("""Argument to convolution operation
                           must be a Tensor object.""")
    if pad:
        l, r, t, b = pad, pad, pad, pad
    l = (other.dims[-1]-1)//2 if not l else l
    r = (other.dims[-1]-1)//2+(other.dims[-1]-1)%2 if not r else r
    t = (other.dims[-2]-1)//2 if not t else t
    b = (other.dims[-2]-1)//2+(other.dims[-2]-1)%2 if not b else b
    array, mask = Tensor.rec_con(self.array, other.array, stride, l, r,t,b)
```

```
    return Tensor(array=array), Tensor(array=mask)

def find_default_convolve_pad(self, other):
    l = (other.dims[-1]-1)//2
    r = (other.dims[-1]-1)//2+(other.dims[-1]-1)%2
    t = (other.dims[-2]-1)//2
    b = (other.dims[-2]-1)//2+(other.dims[-2]-1)%2
    return (l, r, t, b)

def rec_tanh(array, deriv):
    r = []
    if isinstance(array, list):
        for a in array:
            r.append(Tensor.rec_tanh(a, deriv))
    elif deriv:
        r = 1-(math.tanh(array)**2)
    else:
        r = math.tanh(array)
    return r

def tanh(self, deriv=False):
    return Tensor(array=Tensor.rec_tanh(self.array, deriv))

def rec_soft_max(array, deriv):
    r = []
    if isinstance(array[0], list):
        for a in array:
```

```
            r.append(Tensor.rec_soft_max(a, deriv))
        elif deriv:
            raise Exception("FFS")
        else:
            s = 0
            for a in array:
                try:
                    s += math.e**a
                except Exception as e:
                    print(e)
                    print(s)
                    print(a)
            for a in array:
                try:
                    r.append(math.e**a/s)
                except ZeroDivisionError:
                    r.append(0)
                except Exception as e:
                    print(e)
                    print(s)
                    print(a)
        return r

    def soft_max(self, deriv=False):
        return Tensor(array=Tensor.rec_soft_max(self.array, deriv))

    def sigmoid(self, deriv=False):
```

```
    return Tensor(array=Tensor.recursive_sigmoid(self.array, deriv))

def recursive_sigmoid(array, deriv):
    r = []
    if isinstance(array, list):
        for i in array:
            r.append(Tensor.recursive_sigmoid(i, deriv))
    elif deriv:
        r = (1/(1+math.exp(-array)))*(1-(1/(1+math.exp(-array))))
    else:
        try:
            r = 1/(1+math.exp(-array))
        except:
            r = 0
    return r

def rec_transpose(array, first=False):
    result = []
    if isinstance(array, list):
        if isinstance(array[0], list):
            if isinstance(array[0][0], list):
                for sub_array in array:
                    result.append(Tensor.rec_transpose(sub_array[index]))
            else:
                for index in range(len(array[0])):
                    result.append([])
                    for sub_array in array:
```

```
                    result[index].append(sub_array[index])
    if first:
        return Tensor(array=result)
    return result

def transpose(self):
    return Tensor.rec_transpose(self.array, True)

def recursive_one_hot(array, first=False):
    result = []
    if isinstance(array, list):
        if isinstance(array[0], list):
            for sub_array in array:
                result.append(Tensor.recursive_one_hot(sub_array))
        else:
            max_val = -99999999999999
            for val in array:
                max_val = val if val>max_val else max_val
            for val in array:
                result.append(1 if val>=max_val else 0)
                max_val += 1 if val>=max_val else 0
    if first:
        return Tensor(array=result)
    return result

def one_hot(self):
    return Tensor.recursive_one_hot(self.array, True)
```

```
def recursive_add_weights(array):
    if isinstance(array[0], list):
        for i in array:
            Tensor.recursive_add_weights(i)
    else:
        array.append(0)

def add_weights(self):
    Tensor.recursive_add_weights(self.array)
    self.dims[-1] += 1
```

```
import tkinter as Tk
import pickle
from ImmutableEntry import ImmutableEntry
from Tensor import Tensor
from Network import *
from Popup import Popup

def dummy():
    pass

def rgb(r, g, b):
    r = str(hex(max(min(int(r), 255), 0)))[2:]
    g = str(hex(max(min(int(g), 255), 0)))[2:]
    b = str(hex(max(min(int(b), 255), 0)))[2:]
    r = r if len(r)==2 else r+"0"
    g = g if len(g)==2 else g+"0"
    b = b if len(b)==2 else b+"0"
    return "#{0}{1}{2}".format(r, g, b)


class MyFrame(Tk.Frame):

    def clear(self):
        for widget in self.winfo_children():
```

```
            widget.destroy()


class TensorVisualisor(MyFrame):

    def __init__(self, parent, settings):
        Tk.Frame.__init__(self, parent)
        self.canvas = Tk.Canvas(self)
        self.canvas.place(relx=0, rely=0.2, relwidth=1, relheight=0.8)
        self.config(borderwidth=4, relief="raised")
        self.dim_buttons = []
        self.linked = []
        self.current_rec = False
        self.settings = settings
        self.settings.tensor_visualisors.append(self)

    def destroy(self):
        self.settings.tensor_visualisors.remove(self)
        Tk.Frame.destroy(self)

    def clear(self):
        for widget in self.winfo_children():
            if widget!=self.canvas:
                widget.destroy()
        self.canvas.delete("all")

    def display(self, array, highlight_max=None):
```

```
    self.update()
    if highlight_max is not None:
        highlight_max = max(max(array))
    self.update()
    h = self.canvas.canvasx(self.canvas.winfo_width())//len(array[0])
    w = self.canvas.canvasy(self.canvas.winfo_height())//len(array)
    self.h, self.w = h, w
    for i, a in enumerate(array):
        for j, b in enumerate(a):
            coords = (j*h, i*w, (j+1)*h, (i+1)*w)
            yellow = rgb(255, 255, 0)
            test = b==highlight_max and highlight_max is not None
            fill = yellow if test else rgb(b*256, 0, -b*256)
            self.canvas.create_rectangle(coords, fill=fill)
            if self.settings.numbers:
                fill = rgb(0, 255, 255)
                t = str(b)[:3] if b>0 else str(b)[:4]
                coords = ((coords[0]+coords[2])/2, (coords[1]+coords[3])/2)
                self.canvas.create_text(coords, fill=fill, text=t)

def refresh(self):
    self.visualise(self.tensor, self.indices)

def remove_last_rec(self):
    if self.current_rec:
        self.canvas.delete(self.current_rec)
        self.current_rec = False
```

```
def visualise(self, tensor, indices=False, highlight_max=None):
    self.tensor = tensor
    self.clear()
    if len(tensor.dims)<3:
        self.indices = False
        self.display(tensor.array, highlight_max)
    else:
        self.indices = [0]*(len(tensor.dims)-2) if not indices else indices
        to_display = tensor.array
        for i in self.indices:
            to_display = to_display[i]
        self.display(to_display)
        self.display_dimension_options(tensor)
    self.update()

def link(self, linked, fire_index, call_index):
    self.linked.append([linked, fire_index, call_index])

def animate_max_pool(self, window_size, position):
    if self.current_rec:
        self.canvas.delete(self.current_rec)
    w, p = window_size, position
    self.current_rec = self.canvas.create_rectangle(p[0]*self.h,
                        p[1]*self.w, (p[0]+w)*self.h, (p[1]+w)*self.w,
                        outline=rgb(255,255,0))
    array = self.tensor.array
```

```
    for index in self.indices:
        array = array[index]
    result = []
    for i in range(window_size):
        row = []
        for j in range(window_size):
            row.append(array[p[1]+i][p[0]+j])
        result.append(row)
    return result

def move(self, index, increment, linked=False):
    self.indices[index] += increment
    self.indices[index] = min(max(0, self.indices[index]),
                              self.tensor.dims[index]-1)
    self.visualise(self.tensor, self.indices)
    if not linked:
        for linked, fire_index, call_index in self.linked:
            if index==fire_index:
                linked.move(call_index, increment, True)

def display_dimension_options(self, tensor):
    text = "Tensor has {0} spacial dimensions.".format(len(tensor.dims))
    label = Tk.Label(self, text=text)
    label.place(relx=0.02, rely=0.01, relwidth=0.96, relheight=0.06)
    for i, index in enumerate(self.indices):
        c0 = lambda i=i, b=-1: self.move(i, b)
        c1 = lambda i=i, b=1: self.move(i, b)
```

```
            t = "{0} dimension".format(2+len(self.indices)-i)
            l0 = Tk.Label(self, text=t)
            b0 = Tk.Button(self, text="-", command=c0)
            l1 = Tk.Label(self, text=str(index))
            b1 = Tk.Button(self, text="+", command=c1)
            l0.place(relx=i*0.2+0.1, rely=0.07, relwidth=0.15, relheight=0.045)
            b0.place(relx=i*0.2+0.1, rely=0.13, relwidth=0.05, relheight=0.06)
            l1.place(relx=i*0.2+0.15, rely=0.13, relwidth=0.05, relheight=0.06)
            b1.place(relx=i*0.2+0.2, rely=0.13, relwidth=0.05, relheight=0.06)
            self.dim_buttons.append([b0, b1])


class ConvolutionVisualisor(MyFrame):

    def __init__(self, parent, settings):
        Tk.Frame.__init__(self, parent, borderwidth=4, relief="raised")
        self.parent = parent
        self.settings = settings
        self.first = True
        self.par_res_vis, self.sel_vis = False, False
        self.ani_but = Tk.Button(self, text="Animate", command=self.pause)
        self.ani_but.place(relx=0.07, rely=0.03, relwidth=0.06, relheight=0.04)
        b = Tk.Button(self, text="faster", command=self.faster)
        b.place(relx=0.13, rely=0.03, relwidth=0.06, relheight=0.04)
        b = Tk.Button(self, text="slower", command=self.slower)
        b.place(relx=0.01, rely=0.03, relwidth=0.06, relheight=0.04)
        self.time_div = 1
```

```
        self.pause_var = True
        self.row, self.col = 0, 0
        self.show_info()

    def pause(self):
        self.pause_var = not self.pause_var
        if not self.pause_var:
            self.ani_but.config(text="Pause")
            self.animate(self.row, self.col)
        else:
            self.ani_but.config(text="Resume")

    def show_info(self):
        t = Tk.Text(self)
        text = """During animation the currently selected subsection of the input tensor will
be highlighted.
The currently selected section will be expanded and shown above the kernel.
The result of the cross product will be shown below the kernel.
The value this operation coresponds to in the output tensor will be highlighted
The sum of the cross-product is shown to the left, this number then passes through the
activiation function before being placed in the output tensor."""
        t.insert("0.0", text)
        t.place(relx=0.6, rely=0, relwidth=0.4, relheight=0.2)
        self.l = Tk.Label(self, text="Sum of cross-product:  ")
        self.l.place(relx=0.1, rely=0.1, relwidth=0.3, relheight=0.05)
```

```
def faster(self):
    self.time_div *= 2
    self.time_div = min(self.time_div, 256)

def slower(self):
    self.time_div /= 2
    self.time_div = max(1/32, self.time_div)

def link_visualisors(self, img_vis, ker_vis, res_vis):
    ker_vis.link(res_vis, 0, 1)
    res_vis.link(ker_vis, 1, 0)
    for i in range(len(img_vis.tensor.dims)-2):
        img_vis.link(res_vis, i, i)
        res_vis.link(img_vis, i, i)

def visualise(self, image, kernel, result):
    self.image = image
    self.kernel = kernel
    self.result = result
    img_vis = TensorVisualisor(self, self.settings)
    ker_vis = TensorVisualisor(self, self.settings)
    res_vis = TensorVisualisor(self, self.settings)
    img_vis.place(relx=0, rely=0.2, relwidth=0.4, relheight=0.8)
    ker_vis.place(relx=0.4, rely=0.38, relwidth=0.2, relheight=0.28)
    res_vis.place(relx=0.6, rely=0.2, relwidth=0.4, relheight=0.8)
    img_vis.visualise(image)
    ker_vis.visualise(kernel)
```

```
        res_vis.visualise(result)
        self.link_visualisors(img_vis, ker_vis, res_vis)
        self.img_vis = img_vis
        self.res_vis = res_vis
        self.ker_vis = ker_vis


    def destroy_padding_popup(self):
        self.p.destroy()
        l, r, t, b = self.image.find_default_convolve_pad(self.kernel)
        test = self.image.zero_pad(l, r, t, b)
        self.img_vis.visualise(test)
        self.animate()

    def explain_padding(self):
        zero_pad = self.image.find_default_convolve_pad(self.kernel)
        self.p = Popup(self.parent, 0.2, 0.2, 0.6, 0.6, "Zero Padding Applied")
        self.p.display_message([
            "Zero padding is first applied to the input tensor.",
"""with {0} columns of 0's on the left,
{1} on the right, {2} rows of 0's on the top and {3} on the bottom
press""".format(zero_pad[0], zero_pad[1], zero_pad[2], zero_pad[3]),
"Close and observe that this padding has been applied to the input tensor"])
        b = Tk.Button(self.p, text="Close", command=self.destroy_padding_popup)
        b.grid(row=3, column=0, sticky="nsew")
```

```
def animate(self, row=0, col=0):
    if self.first:
        self.explain_padding()
        l = Tk.Label(self, text="Current Selection")
        l.place(relx=0.4, rely=0, relwidth=0.2, relheight=0.05)
        l = Tk.Label(self, text="Kernel")
        l.place(relx=0.4, rely=0.33, relwidth=0.2, relheight=0.05)
        l = Tk.Label(self, text="Cross Product")
        l.place(relx=0.4, rely=0.66, relwidth=0.2, relheight=0.05)
        self.first = False
        return
    if not self.pause_var:
        if self.par_res_vis:
            self.par_res_vis.destroy()
            self.sel_vis.destroy()
        window_size = self.kernel.dims[-1]
        selection = self.img_vis.animate_max_pool(window_size, (row, col))
        self.res_vis.animate_max_pool(1, (row, col))
        selection = Tensor(array=selection)
        sel_vis = TensorVisualisor(self, self.settings)
        sel_vis.place(relx=0.4, rely=0.05, relwidth=0.2, relheight=0.28)

        sel_vis.visualise(selection)
        filt = self.kernel.array
        for index in self.ker_vis.indices:
            filt = filt[index]
        filt = Tensor(array=filt)
```

```
            par_res = selection*filt
            par_res_vis = TensorVisualisor(self, self.settings)
            par_res_vis.place(relx=0.4, rely=0.71, relwidth=0.2, relheight=0.28)
            par_res_vis.visualise(par_res)
            s = 0
            for i in par_res.array:
                for j in i:
                    s += j
            self.l.config(text="Sum of cross-product: "+str(s))
            self.par_res_vis, self.sel_vis = self.par_res_vis, self.sel_vis
            row = row+1 if row<=self.result.dims[-1]-2 else 0
            col = col+1 if row==0 else col
            self.row, self.col = row, col
            if col<=self.result.dims[-2]-2:
                self.parent.after(int(1000/self.time_div),
                                  lambda r=row, c=col: self.animate(r, c))
            else:
                self.par_res_vis.destroy()
                self.sel_vis.destroy()
                self.img_vis.remove_last_rec()
                self.res_vis.remove_last_rec()
                self.row, self.col = 0, 0


class MaxPoolVisualisor(MyFrame):
```

```
def __init__(self, parent, settings):
    Tk.Frame.__init__(self, parent, borderwidth=4, relief="raised")
    l = Tk.Label(self, text = """During animation the selection will
                            be highlighted in yellow""")
    l.place(relx=0.61, relheight=0.05, relwidth=0.36, rely=0.05)
    l = Tk.Label(self, text = """as will the corresponding
                            value in the output""")
    l.place(relx=0.61, relheight=0.07, relwidth=0.36, rely=0.1)
    l = Tk.Label(self, text = """below the selection is shown with the
                            highest value highlighted in yellow""")
    l.place(relx=0.61, relheight=0.07, relwidth=0.36, rely=0.15)
    self.settings = settings
    self.parent = parent
    self.b = Tk.Button(self, text="Animate", command=self.pause)
    self.b.place(relx=0.15, rely=0.05, relwidth=0.2, relheight=0.1)
    b = Tk.Button(self, text="Faster", command=self.faster)
    b.place(relx=0.36, rely=0.05, relwidth=0.14, relheight=0.1)
    b = Tk.Button(self, text="Slower", command=self.slower)
    b.place(relx=0, rely=0.05, relwidth=0.14, relheight=0.1)
    b = Tk.Button(self, text="Stop", command=self.stop)
    b.place(relx=0.51, rely=0.05, relwidth=0.09, relheight=0.1)
    self.selected_vis, self.pause_var, self.stopped = False, True, False
    self.row, self.col = 0, 0
    self.time_div = 1

def stop(self):
    self.row, self.col = 0, 0
```

```
        self.stopped = True
        try:
            self.selected_vis.destroy()
        except:
            pass
        self.pause_var = True
        self.b.config(text="Animate")
        self.large_vis.remove_last_rec()
        self.small_vis.remove_last_rec()

    def faster(self):
        self.time_div *= 2
        self.time_div = min(self.time_div, 256)

    def slower(self):
        self.time_div /= 2
        self.time_div = max(1/32, self.time_div)

    def pause(self):
        self.pause_var = not self.pause_var
        text = self.b.config()["text"][4]
        text = "Resume" if text=="Pause" else "Pause"
        self.b.config(text=text)
        if not self.pause_var:
            self.animate(self.row, self.col)

    def animate(self, row=0, col=0):
```

```
        if not self.pause_var:
            if self.selected_vis:
                self.selected_vis.destroy()
            self.selected_vis = TensorVisualisor(self, self.settings)
            self.selected_vis.place(relx=0.67, rely=0.2, relwidth=0.26,
                                    relheight=0.26)
            array = self.large_vis.animate_max_pool(self.window_size, (row, col))
            self.selected_vis.visualise(Tensor(array=array), highlight_max=True)
            self.small_vis.animate_max_pool(1, (int(row/self.stride),
                                                int(col/self.stride)))
            row = 0 if self.large.dims[-2]-self.stride == row else row+self.stride
            col = col if row != 0 else col+self.stride
            if col <= self.large.dims[-1]-self.stride:
                self.parent.after(int(1000/self.time_div),
                                  lambda r=row, c=col: self.animate(r, c))
            else:
                self.stop()
        if not self.stopped:
            self.row, self.col = row, col
        else:
            self.stopped = False


    def link_visualisors(self):
        for i in range(len(self.large_vis.tensor.dims)-2):
            self.large_vis.link(self.small_vis, i, i)
            self.small_vis.link(self.large_vis, i, i)
```

```
    def visualise(self, large, small, window_size, stride):
        self.large, self.small = large, small
        self.window_size, self.stride = window_size, stride
        large_vis = TensorVisualisor(self, self.settings)
        small_vis = TensorVisualisor(self, self.settings)
        large_vis.place(relx=0, rely=0.2, relwidth=0.6, relheight=0.8)
        small_vis.place(relx=0.6, rely=0.465, relwidth=0.4, relheight=0.8*(2/3))
        large_vis.visualise(large)
        small_vis.visualise(small)
        self.large_vis, self.small_vis = large_vis, small_vis
        self.link_visualisors()


class NetworkVisualisorSettings(MyFrame):

    def __init__(self, parent):
        Tk.Frame.__init__(self, parent, borderwidth=4, relief="raised")
        self.tensor_visualisors = []
        self.numbers = False
        b = Tk.Button(self, text="Show Numbers", fg="red")
        b.config(command=lambda b=b: self.invert_numbers(b))
        b.place(relx=0, relwidth=0.1, rely=0, relheight=1)

    def invert_numbers(self, b):
        self.numbers = not self.numbers
        b.config(fg="green" if self.numbers else "red")
```

```
        for tensor_visualisor in self.tensor_visualisors:
            tensor_visualisor.refresh()


class NetworkVisualisor(Tk.Tk):

    def __init__(self, network=False):
        Tk.Tk.__init__(self)
        self.layer_buttons, self.conv_buttons, self.max_buttons = [], [], []
        self.settings = NetworkVisualisorSettings(self)
        self.settings.place(relx=0.1, rely=0.1, relwidth=0.8, relheight=0.1)
        self.vis = None
        self.net = network
        self.geometry("1400x850")
        if network:
            self.visualise()

    def display_convolution(self, image, kernel, result):
        if self.vis is not None:
            self.vis.destroy()
        self.vis = ConvolutionVisualisor(self, self.settings)
        self.vis.place(relx=0.02, rely=0.2, relwidth=0.96, relheight=0.8)
        self.vis.visualise(image, kernel, result)

    def display_tensor(self, tensor):
        if self.vis is not None:
            self.vis.destroy()
```

```
        self.vis = TensorVisualisor(self, self.settings)
        self.vis.place(relx=0.2, rely=0.2, relwidth=0.6, relheight=0.8)
        self.vis.visualise(tensor)

    def display_layer(self, layer, c, c0):
        if isinstance(layer, list):
            t = "Output" if c0+2==len(self.net.layers) else "Hidden"
            b1 = Tk.Button(self, text=t)
            b1.place(x=10+120*c, rely=0, width=120, relheight=0.05)
            b = Tk.Button(self, text=str(layer))
            b.place(x=10+120*c, rely=0.05, width=120, relheight=0.05)
            self.layer_buttons.append((b, b1))
        else:
            if layer.layer_type == "max_pool":
                com = dummy
            else:
                com = lambda tensor=layer.tensor: self.display_tensor(tensor)
            test = layer.layer_type != "max_pool"
            t = str(layer.tensor.dims) if test else "N/A"
            b = Tk.Button(self, text=t, command=com)
            b1 = Tk.Button(self, text=layer.layer_type)
            b.place(x=10+120*c, rely=0, width=120, relheight=0.05)
            b1.place(x=10+120*c, rely=0.05, width=120, relheight=0.05)
            if layer.layer_type == "convolutional":
                self.conv_buttons.append(b1)
            elif layer.layer_type == "max_pool":
                self.max_buttons.append(b1)
```

```
        elif layer.layer_type == "input":
            b1.config(command=lambda t=layer.tensor:self.display_tensor(t))

def display_max_pool(self, large, small, window_size, stride):
    if self.vis is not None:
        self.vis.destroy()
    self.vis = MaxPoolVisualisor(self, self.settings)
    self.vis.place(relx=0.02, rely=0.2, relwidth=0.96, relheight=0.8)
    self.vis.visualise(large, small, window_size, stride)


def forward_pass(self, tensor):
    self.net.forward_pass(Tensor(array=tensor[:50]))
    it = zip(self.layer_buttons, self.net.pass_results[1:])
    for (b0, b1), tensor in it:
        b0.config(command=lambda x=tensor: self.display_tensor(x))
        b1.config(command=lambda x=tensor: self.display_tensor(x))
    it = zip(self.conv_buttons, self.net.convolutional_layers())
    for button, (index, kernel) in it:
        image = self.net.pass_results[index-1]
        result = self.net.pass_results[index]
        button.config(command = lambda i=image, k=kernel, r=result:
                      self.display_convolution(i, k, r))
    it = zip(self.max_buttons, self.net.max_layers())
    for button, index in it:
        large = self.net.pass_results[index-1]
        small = self.net.pass_results[index]
```

```
            window_size = self.net.layers[index].window_size
            stride = self.net.layers[index].stride
            button.config(command = lambda l=large, s=small, w=window_size,
                          st=stride: self.display_max_pool(l, s, w, st))

    def visualise(self, network=False):
        i = 0
        i0 = 0
        self.net = network if network else self.net
        for layer in self.net.layers:
            self.display_layer(layer, i, i0)
            i += 1
            if layer.layer_type != "input":
                self.display_layer(layer.result_dims, i, i0)
                i += 1
                i0 += 1

network = Network()
path = "/Users/user/Desktop/Tensor/"
images = pickle.load(open("{0}test_images.pickle".format(path), "rb"))
labels = pickle.load(open("{0}test_labels.pickle".format(path), "rb"))
print("LOADING NETWORK")
network.load(76.0)
print("NETWORK LOADED")

visualisor = NetworkVisualisor(network)
visualisor.forward_pass(images)
```

```
import pickle
from Network import *

filters = [[
 [-1 ,-1 ,-1 ,-1 ,-1 ],
 [0.6,0.6,0.6,0.6,0.6],
 [1  ,1,  1,  1,  1, ],
 [0.6,0.6,0.6,0.6,0.6],
 [-1 ,-1 ,-1 ,-1 ,-1 ],
],[
 [-1,0.6,1,0.6,-1],
 [-1,0.6,1,0.6,-1],
 [-1,0.6,1,0.6,-1],
 [-1,0.6,1,0.6,-1],
 [-1,0.6,1,0.6,-1],
 ],[
 [1  ,0.6,-1 ,-1 ,-1 ],
 [0.6,1  ,0.6 ,-1 ,-1],
 [-1 ,0.6,1  ,0.6,-1 ],
 [-1 ,-1 ,0.6,1  ,0.6],
 [-1 ,-1 ,-1,0.6 ,1  ],
 ],[
 [-1 ,-1 ,-1 ,0.6,1  ],
```

```
 [-1 ,-1 ,0.6,1  ,0.6],
 [-1 ,0.6,1  ,0.6,-1 ],
 [0.6,1  ,0.6,1  ,-1 ],
 [1  ,0.6,-1 ,-1 ,-1 ],
 ]
]


if __name__=="__main__":
    network = Network(0.6)
    images = pickle.load(open("training_images.pickle", "rb"))
    labels = pickle.load(open("training_labels.pickle", "rb"))
    test_images = pickle.load(open("test_images.pickle", "rb"))
    test_labels = pickle.load(open("test_labels.pickle", "rb"))
    for index, image in enumerate(images):
        images[index] = image[:-1]
    image_tensor = Tensor(array=images[:25])
    l0 = InputLayer(image_tensor, reshape=[28, 28])
    l1 = ConvolutionalLayer(5, 4, l0, pretrained=filters)
    l2 = MaxPoolLayer(l1)
    l3 = FullyConnectedLayer(120, l2, drop_out=20)
    l4 = FullyConnectedLayer(80, l3, drop_out=20)
    l5 = OutputLayer(10, l4)
    for l in [l0, l1, l2, l3, l4, l5]:
        network.add_layer(l)
    network.finalise()
    score = 0
```

```
while True:
    print("TESTING...", end="")
    score = network.test(test_images, test_labels, 100)
    print("Score: {0}%".format(score))
    network.train(images, labels, (1-(100-score)/100), 4, 100)
    print()
    network.save(score)
```