VIDEO "Neural Differential Equations"
— by "Siraj Raval"

Closing Summary of Neural Ordinary Differential Equations

- A neural network is a popular type of machine learning model
- Neural Networks are built with linear algebra and optimized using Calculus
- Neural networks consist of a series of "layers", which are just matrix operations
- Each layer introduces a little bit of error that compounds through the network
- The way to reduce that error is to add more and more layers
- The problem is that we see a drop off in performance after a certain # of layers
- A solution to this was proposed by Microsoft for the 2015 ImageNet competiton (residual networks)
- Residual Networks connect the output of previous layers to the output of new layers
- Prof. Duvenaud's team at University of Toronto noticed that that ResNets are similar to a primative "Ordinary Differential Equation" Solver called "Euler's Method"
- Ordinary Differential Equations involve one or more ordinary derivatives of unkown functions. 1 independent variable.
- Partial Differential Equations involve one or more partial derivatives of unknown functions. 2 or more independnet variables.
- Eulers method is a numerical method to sovle 1st order differential equations
- More efficient than Eulers method is the adjoint method. And this acts as our optimization strategy
- The result? No need to specificy # of layers beforehand, now just specifiy accuracy. it will train itself.
- No more discrete layers, instead a continous computational block

Applications -Irregular time series data (medical history recorded at random times) , discrete layers are bad at this -Memory Efficiency (constant memory), slower training time, faster test time,

BENEFITS of **NEURALODE** (vs. neural nets.)
- time series / physical models
- density modeling !!
- supervised learning (computations)

generative density model ?

Example: ECG **classification** (MIT dataset)

ResNet accuracy: 0.974
ODENet accuracy: 0.976

ResNet #params: 182.853
ODENet #params: 59.333
(30% of the params of ResNet)

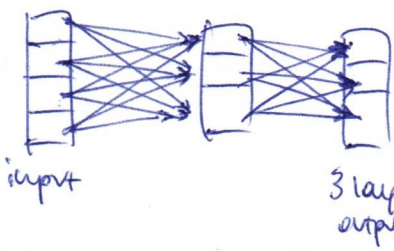
time series data (anything depending on time)

→ Neural ODE > recurrent neural net
(e.g. residual / dense / .. networks)

neural networks = functions approximators

↓

have layers:



discrete number of layers, what if we want a continuous amount of layer?

why →

- time series data
- if we frame it as an ODE, we can use different optimizers to train the network (better than gradient descent)

→ differential equation solvers

neural nets approx. functions

↓

the more layer the better the approx.? NO.

↓

ResNet: "it invece"

$x_{k+1} = F(x_k) + x_k$

wait a minute!

ODE

dalla derivata vogliamo capire la funzione

↓

Eulero: "I got this" (±)

$x(t+h) = hx'(t) + x(t)$

$x(t+h) = h \cdot f(x) + x(t)$

How do we backpropagate the error?

→ we need something that combines backpropagation and ODE

⇒ adjoint method

IlSourcell Add files via upload History

1 contributor

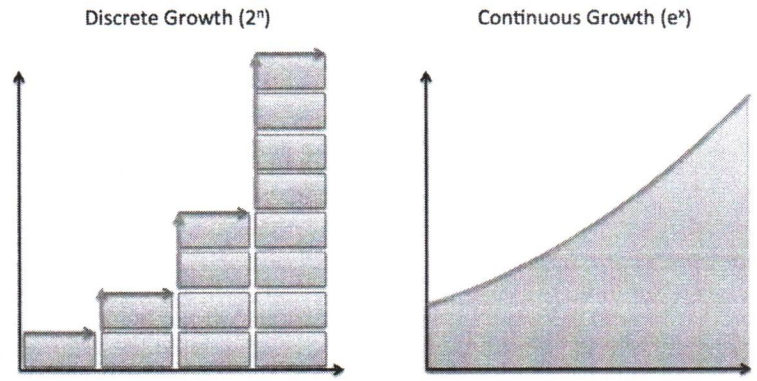
846 lines (846 sloc) 52.8 KB ...

In [0]:
```
%%HTML
<video controls loop width="1000" height="600" controls>
  <source src="https://video.twimg.com/tweet_video/Dr_CCsCUUAAwRS1.mp4" type="video/mp4">
</video>
```

Neural Ordinary Differential Equations

Summary

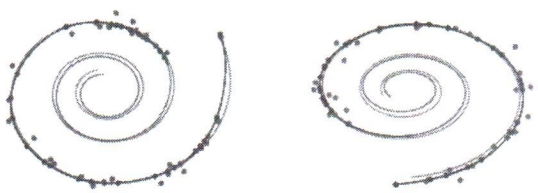
NeurIPS is the largest AI conference in the world. 4,854 papers were submitted. 4 received "Best paper" award. This is one of them. The basic idea is that neural networks are made up of stacked layers of simple computation nodes that work together to approximate a function. If we re-frame a neural network as an "Ordinary Differential Equation", we can use existing ODE solvers (like Euler's method) to approximate a function. This means no discrete layers, instead the network is a continous function. No more specifying the # of layers beforehand, instead specify the desired accuracy, it will learn how to train itself within that margin of error. It's still early stages, but this could be as big a breakthrough as GANs!



Demo

An ODENet approximated this spiral function better than a Recurrent Network.

(a) Recurrent Neural Network



(b) Latent Neural Ordinary Differential Equation

Why Does this matter?

1. Faster testing time than recurrent networks, but slower training time. Perfect for low power edge computing! (precision vs speed)
2. More accurate results for time series predictions (!!) i.e continous-time models
3. Opens up a whole new realm of mathematics for optimizing neural networks (Diff Equation Solvers, 100+ years of theory) 4, Compute gradients with constant memory cost

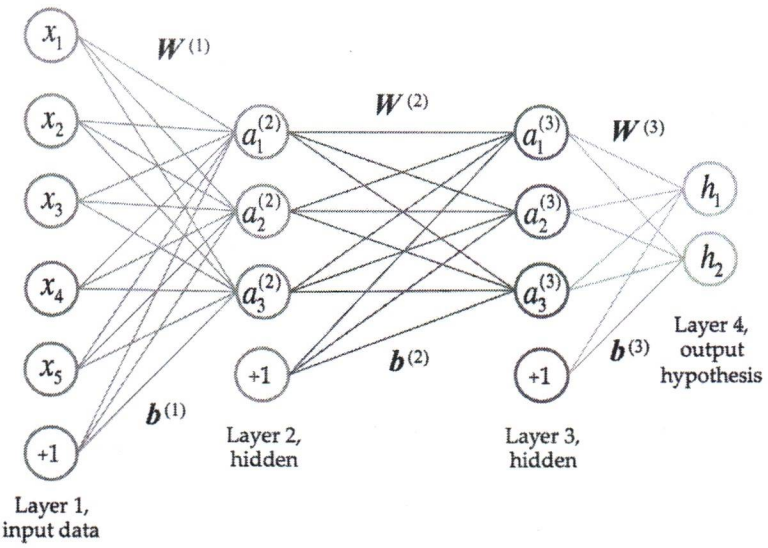
Concepts we'll learn about in this video

1. Basic neural network theory
2. "Residual" neural network theory
3. Ordinary Differential Equations (ODEs)
4. ODE Networks
5. Euler's Method to Optimize an ODENet
6. Adjoint Method for ODENet Optimization
7. ODENet's Applied to time series data
8. Future Applications of ODENets

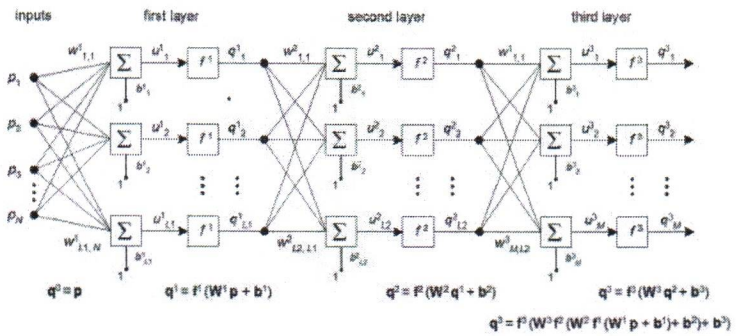
1 Basic Neural Network Theory

- Neural Networks are a popular type of ML model
- Neural Networks are built with linear algebra (matrices & matrix operations) & optimized using Calculus (gradient descent & other algorithms)
- Neural networks consist of a series of "layers", which are just matrix operations
- Each layer introduces a little bit of error that compounds through the network

Basic Neural Network Diagram



More Detailed Neural Network Diagram



a neural network is a giant composite function

Basic Neural Network Example

```
In [0]: import numpy as np

        # compute sigmoid nonlinearity
        def sigmoid(x):
            output = 1/(1+np.exp(-x))
            return output

        # convert output of sigmoid function to its derivative
        def sigmoid_output_to_derivative(output):
            return output*(1-output)

        # input dataset
        X = np.array([  [0,1],
                        [0,1],
                        [1,0],
                        [1,0] ])

        # output dataset
        y = np.array([[0,0,1,1]]).T

        # seed random numbers to make calculation
        # deterministic (just a good practice)
        np.random.seed(1)

        # initialize weights randomly with mean 0
        synapse_0 = 2*np.random.random((2,1)) - 1

        for iter in range(10000):

            # forward propagation
            layer_0 = X
            layer_1 = sigmoid(np.dot(layer_0,synapse_0))

            # how much did we miss?
            layer_1_error = layer_1 - y

            # multiply how much we missed by the
```

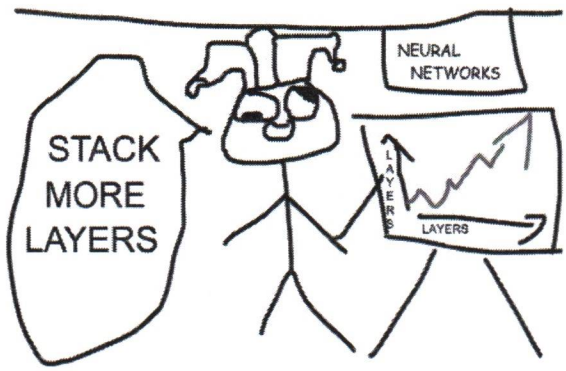
we compute the partial derivative of the error w.r.t. the weights and we propagate backwards

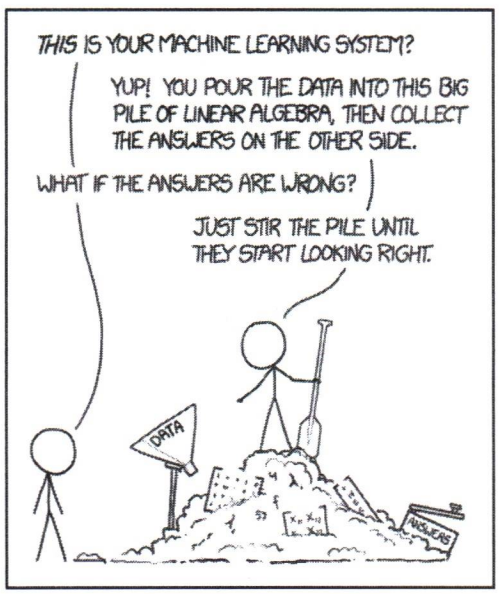
```
e(layer_1)
    synapse_0_derivative = np.dot(layer_0.T,layer_1_delta)

    # update weights
    synapse_0 -= synapse_0_derivative

print("Output After Training:")
print(layer_1)
```

```
Output After Training:
[[0.00505119]
 [0.00505119]
 [0.99494905]
 [0.99494905]]
```

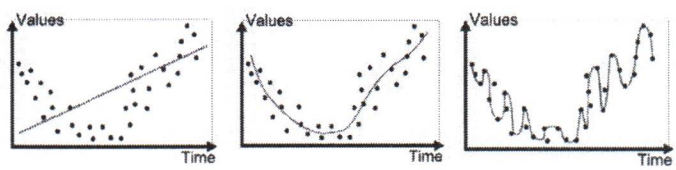
Stack More Layers?

- To reduce compounded error, add more layers!





- The # of layers & # of neurons deeply affec the output of the network
- Too few layers could cause underfitting
- Too many layers coudl cause overfitting + long training time



Underfitted | Good Fit/Robust | Overfitted

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer

Example: Long-Short Term Memory Neural Network with many layers

In [0]:
```
from keras.models import Sequential
```

```
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10

# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a
sequence of vectors of dimension 32
model.add(LSTM(32, return_sequences=True))  # returns a sequenc
e of vectors of dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32



model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# Generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

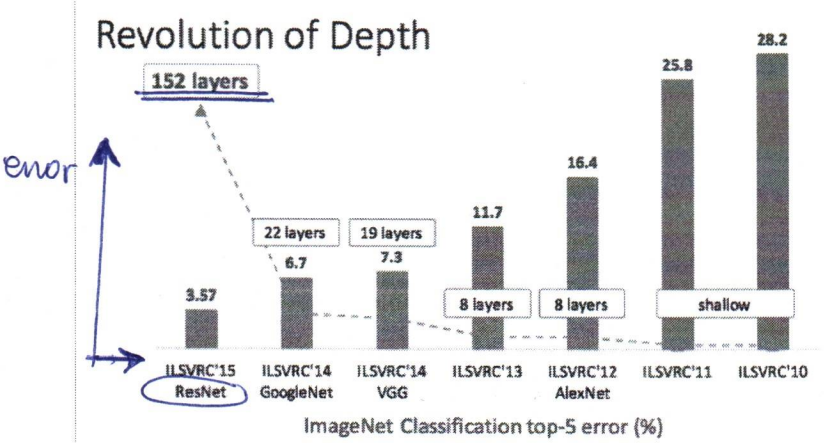
we can add 00 layers, however there's a moment where it doesn't help anymore

Solution

2 Residual Neural Network Theory

A solution to this was proposed by Microsoft for the 2015 ImageNet competiton (residual networks)

- In December of 2015, Microsoft proposed "Residual networks" as a solution to the ImageNet Classification Competition
- ResNets had the best accuracy in the competition
- ResNets utilize "skip-connections" between layers, which increases accuracy.
- They were able to train networks of up to 1000 layers deep while avoiding vanishing gradients (lower accuracy)
- 6 months later, their publicatio already had more than 200 references.



error

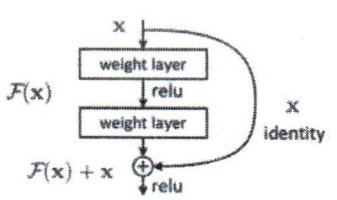
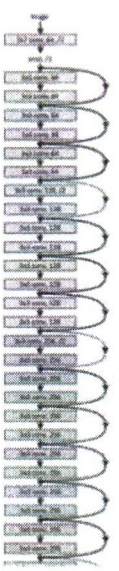
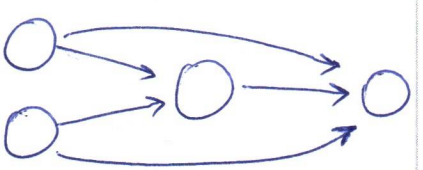


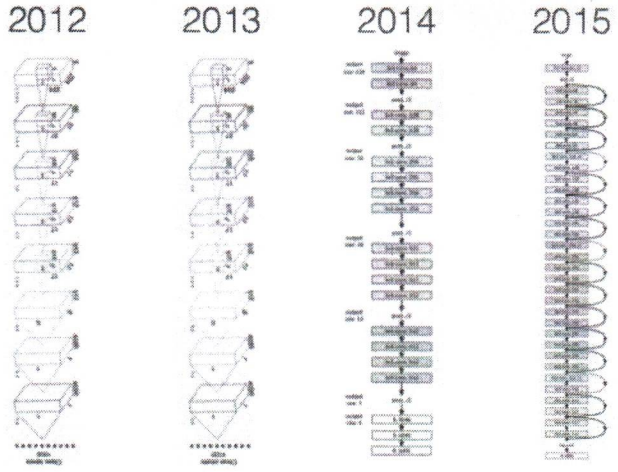
Figure 2. Residual learning: a building block.

Network can decide how deep it needs to be...



ResNet = skip connections:





How do ResNets work?

- Instead of hoping each stack of layers directly fits a desired underlying mapping, we explicitly let these layers fit a residual mapping.
- The original mapping is recast into F(x)+x.
- Residual neural networks do this by utilizing skip connections or short-cuts to jump over some layers.
- The residual layer adds the output of the activation function to the input of the layer.
- This seemingly minor change has led to a rethinking of how neural network layers are designed.
- In its limit as ResNets it will only skip over a single layer
- With an additional weight matrix to learn the skip weights it is referred to as HighwayNets
- With several parallel skips it is referred to as DenseNets

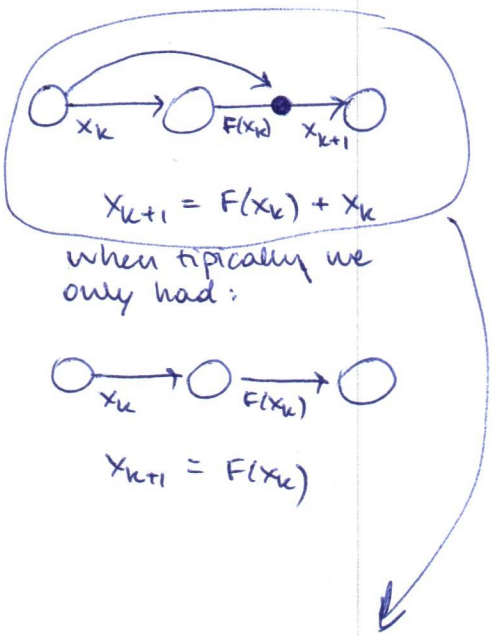
The residual layer is actually quite simple: add the output of the activation function to the original input to the layer. As a formula, the k+1th layer has the formula:

$$x_{k+1} = x_k + F(x_k)$$

where F is the function of the kth layer and its activation. For example, F might represent a convolutional layer with a relu activation. This simple formula is a special case of the formula:

$$x_{k+1} = x_k + hF(x_k),$$

which is the formula for the Euler method for solving ordinary differential equations (ODEs) when h=1



$F(x_k)$ is the output of the second layer, $F(x_k) + x_k$ is the input of the third layer

Wait, WTF is Euler's method? What does differential equations have to do with anything? Hold that thought, look at this code first.

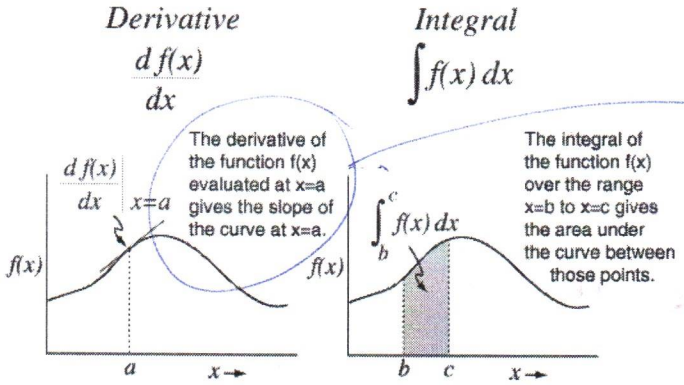
In [0]:
```
#normal convolutional layer

def Unit(x,filters):

    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[
1, 1], padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[
1, 1], padding="same")(out)

    return out
```

In [0]:
```
#residual convolutional layer

def Unit(x,filters):
    res = x
    out = BatchNormalization()(x)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[
1, 1], padding="same")(out)

    out = BatchNormalization()(out)
    out = Activation("relu")(out)
    out = Conv2D(filters=filters, kernel_size=[3, 3], strides=[
1, 1], padding="same")(out)

    out = keras.layers.add([res,out])

    return out
```

between residual networks and Ordinary Differential Equations?

3 Ordinary Differential Equations

- A "differential equation" is an equation that just tells us the slope without specifying the original function whose derivative we are taking

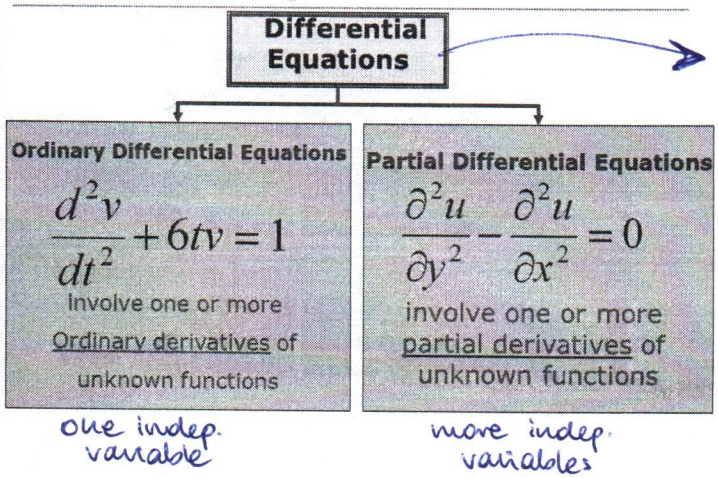


why does it matter? if we derive at any point we obtain the slope and the slope tells us the direction that the function is moving at the specific point in time

What if we have the derivative and we want to obtain the original function? → INTEGRAL

FUNCTION	DERIVATIVE
x^2	$2x$
x^3	$3x^2$
x^4	$4x^3$
x^5	$5x^4$
x^1	$1 \cdot x^0$ (i.e. $1 \cdot 1 = 1$ BECAUSE $x^0 = 1$)
$x^{\frac{1}{2}}$ (i.e. $\sqrt{x}$)	$\frac{1}{2} \cdot x^{-\frac{1}{2}}$ (i.e. $\frac{1}{2\sqrt{x}}$)

Differential Equations



one indep. variable

more indep. variables

the whole point is to find the original function (and neural networks are functions approximators ????!!)

starting from the expression of the derivative

derivative = ...

e.g.

Solve the differential equation: $\frac{dy}{dx} = 5x + 3$

This can be rewritten as:

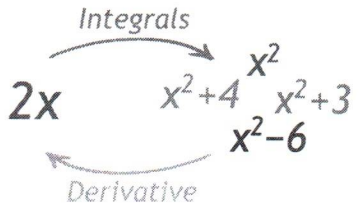
$$\int dy = \int (5x + 3)dx$$

(If you have difficulties with this, think dy/dx as a fraction. It is as if we have multiplied each side by dx)

Now simply integrate both sides:

$$y = 5x^2 + 3x + c$$

alt text

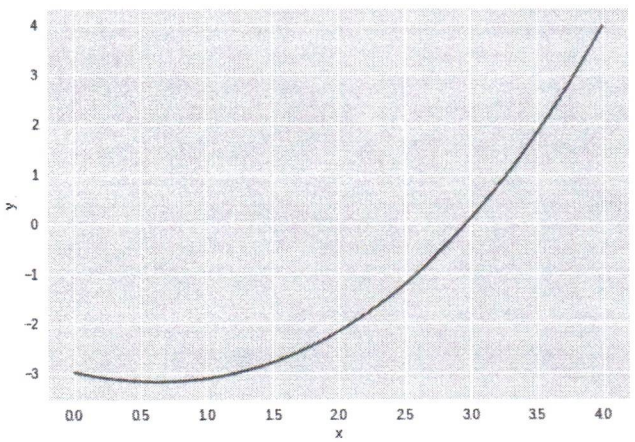


Example Differential Equation

In [0]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# function dy/dx = x + y/5.
func = lambda y,x : x + y/5.
# Initial condition
y0 = -3  # at x=0
# values at which to compute the solution (needs to start at x=0)
x = np.linspace(0, 4, 101)
# solution
y = odeint(func, y0, x)
# plot the solution, note that y is a column vector
plt.plot(x, y[:,0])
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

!!

ODE solver



4 ODE (Ordinary Differential Equation) Networks

Consider a simplified ODE from physics: we want to model the position x of a marble. Assume we can calculate its velocity x' (the derivative of position) at any position x. We know that the marble starts at rest x(0)=0 and that its velocity at time t depends on its position through the formula:

$$x'(t) = f(x)$$

The Euler method solves this problem by following the physical intuition: my position at a time very close to the present depends on my current velocity and position. For example, if you are travelling at a velocity of 5 meters per second, and you travel 1 second, your position changes by 5 meters. If we travel h seconds, we will have travelled 5h meters. As a formula, we said:

$$x(t+h) = x(t) + hx'(t),$$

position at new time = position now + (Δtime) · velocity now

but since we know

$$x'(t) = f(x)$$

we can rewrite this as

$$x(t+h) = x(t) + hf(x).$$

If you squint at this formula for the Euler method, you can see it looks just like the formula for residual layers!

This observation has meant three things for designing neural networks:

- New neural network layers can be created through different numerical approaches to solving ODEs
- The possibility of arbitrarily deep neural networks
- Training of a deep network can be improved by considering the so-called stability of the underlying ODE and its numerical discretization

x(t)

x'

x

where 0 is going to be at time t knowing its starting point and its velocity?

$$\begin{cases} x(0) = 0 & \text{position at } t=0 \\ x'(t) = f(x) & \text{velocity} \end{cases}$$

$\xrightarrow{?}$ x(t)

we frame the residual network equation as a differential equation

⇒ differential equations can represent neural networks

(neural nets can be represented by differential equations)

- To create arbitrarily deep networks with a finite memory footprint, design neural networks based on stable ODEs and numerical discretizations.
- Gradient descent can be viewed as applying Euler's method for solving ordinary differential equation to gradient flow.

$$x'(t) = -\nabla f(x(t))$$

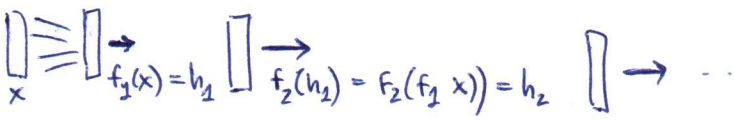
What Does an ODENet Look like?

An ODE is a function that usually describes the change of some system through time. In this setting, time is a continuous variable. Now imagine a neural network is that system, and time is really something more like the depth of the network. Note that there are usually a discrete number of layers in an ANN. This is a notion of continuous number of layers.

- The team didn't use Euler's method, they computed the exact ODE solution (within a small error tolerance) using adaptive solvers (faster)
- The dynamics change smoothly with depth. You can think of this either as having weights that are a function of depth, or as having shared weights across layers but adding the depth as an extra input to f.
- Anywhere you can put a resnet you can put an ODEnet.
- Each ODEBlock can be used to replace a whole stack of ResBlocks.
- In their MNIST example, each ODEBlock replaces 6 ResBlocks.

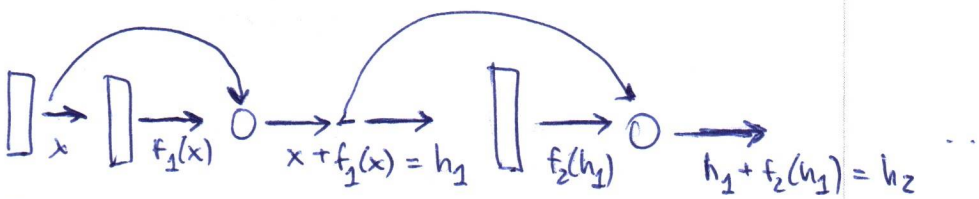
Traditional Deep Nets

```
h1 = f1(x)
h2 = f2(h1)
h3 = f3(h2)
h4 = f3(h3)
y  = f5(h4)
```



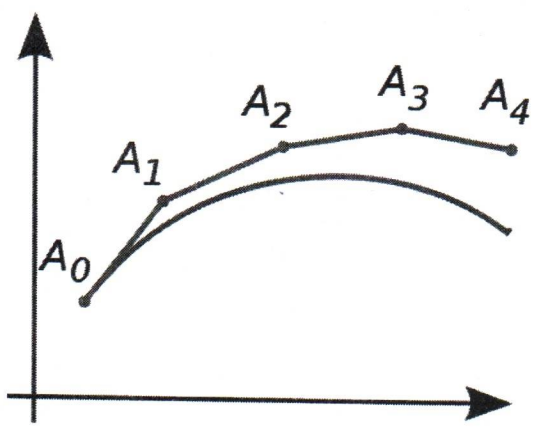
ResNets

```
h1 = f1(x)  + x
h2 = f2(h1) + h1
h3 = f3(h2) + h2
h4 = f4(h3) + h3
y  = f5(h4) + h4
```



- Where f1, f2, etc are neural net layers.
- The idea is that it's easier to model a small change to an almost-correct answer than to output the whole improved answer at once. -This looks like a primitive ODE solver (Euler's method) that solves the trajectory of a system by just taking small steps in the direction of the system dynamics and adding them up. -They connection allows for better training methods.
- What if we define a deep net as a continuously evolving system?
- Instead of updating the hidden units layer by layer, we define their derivative with respect to depth instead
- We can use off-the-shelf adaptive ODE solvers to compute the final state of these dynamics, and call that the output of the neural network.

5 Euler's Method



- We want to recover the blue curve, but all we have is an initial point A_0 (think inputs to the network) and a differential equation.
- From the differential equation, we can calculate the tangent line. If we take a small step along the tangent line, we arrive at A_1, which will be close to the desired blue line if the step is small enough.
- Repeat this process to uncover a polygonal curve A{0}A{1}A{2}...A{n}.

Many neural networks have a composition that looks exactly like the steps of Euler's method. We start with an initial state \mathbf{z}_0, and apply successive transformations over time (layers):

$$\mathbf{z}_1 = \mathbf{z}_0 + f(\mathbf{z}_0, \theta_0)$$

$$\mathbf{z}_3 = \mathbf{z}_2 + f(\mathbf{z}_2, \theta_2)$$

...

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f(\mathbf{z}_t, \theta_t)$$

In the limit, we parameterize the continuous dynamics of hidden units using an ordinary differential equation (ODE) specified by a neural network:

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta)$$

The equivalent of having T layers in the network, is finding the solution to this ODE at time T.

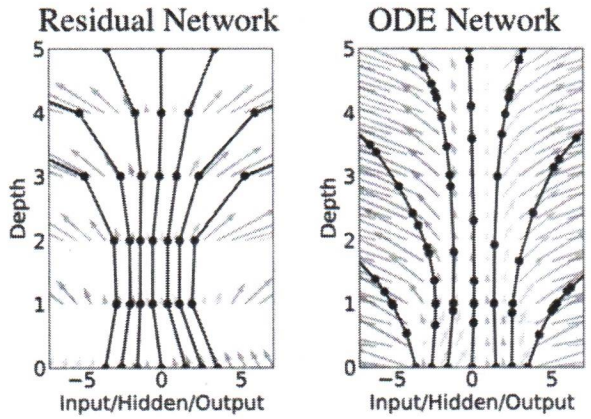


Figure 1: *Left:* A Residual network defines a discrete sequence of finite transformations. *Right:* A ODE network defines a vector field, which continuously transforms the state. *Both:* Circles represent evaluation locations.

- Euler's method is perhaps the simplest method for solving ODEs.
- There since been more than 120 years of development of efficient and accurate ODE solvers.
- Modern ODE solvers provide guarantees about the growth of approximation error, monitor the level of error, and adapt their evaluation strategy on the fly to achieve the requested level of accuracy.
- This allows the cost of evaluating a model to scale with problem complexity.

We've seen how to feed-forward, but how do you efficiently train a network defined as a differential equation? The answer lies in the adjoint method (which dates back to 1962). Think of the adjoint as the instantaneous analog of the chain rule.

6 The Adjoint Method

- This approach computes gradients by solving a second, augmented ODE backwards in time, and is applicable to all ODE solvers.
- This approach scales linearly with problem size, has low memory cost, and explicitly controls numerical error.
- The adjoint captures how the loss function L changes with respect to the hidden state.
- Starting from the output of the network, we can recompute the hidden state backwards in time together with the adjoint.

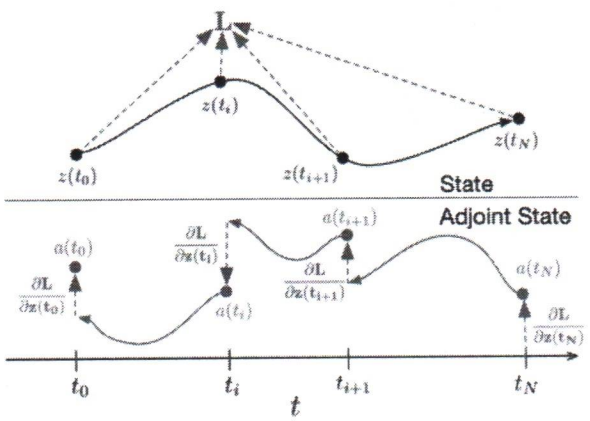


Figure 2: Reverse-mode differentiation through an

ODE solver requires solving an augmented system backwards in time. This adjoint state is updated by the gradient at each observation.

- A third integral then tells us how the loss changes with the parameters \theta (dL/d\theta).
- All three of these integrals can be computed in a single call to an ODE solver, which concatenates the original state, the adjoint, and the other partial derivatives into a single vector.
- Algorithm 1 shows how to construct the necessary dynamics, and call an ODE solver to compute all gradients at once.

Algorithm 1 Reverse-mode derivative of an ODE initial value problem

Input: dynamics parameters θ, start time t_0, stop time t_1, final state $\mathbf{z}(t_1)$, loss gradient $\frac{\partial L}{\partial \mathbf{z}(t_1)}$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial \mathbf{z}(t_1)}^\intercal f(\mathbf{z}(t_1), t_1, \theta)$ ▷ Compute gradient w.r.t. t_1

$s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}, -\frac{\partial L}{\partial t_1}]$ ▷ Define initial augmented state

def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), -, -], t, \theta$): ▷ Define dynamics on augmented state

  return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial \theta}, -\mathbf{a}(t)^\intercal \frac{\partial f}{\partial t}]$ ▷ Concatenate time-derivatives

$[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta)$ ▷ Solve reverse-time ODE

return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷ Return all gradients

ODE Net Example

In [0]:
```
## Import the Adjoint Method (ODE Solver)
from torchdiffeq import odeint_adjoint as odeint

## Normal Residual Block Example

class ResBlock(nn.Module):

    #init a block - Convolve, pool, activate, repeat
    def __init__(self, inplanes, planes, stride=1, downsample=N
one):
        super(ResBlock, self).__init__()
        self.norm1 = norm(inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.norm2 = norm(planes)
        self.conv2 = conv3x3(planes, planes)

    #Forward pass - pass output of one layer to the input of th
e next
    def forward(self, x):
        shortcut = x
        out = self.relu(self.norm1(x))
        out = self.conv1(out)
        out = self.norm2(out)
        out = self.relu(out)
        out = self.conv2(out)

        return out + shortcut

## Ordinary Differential Equation Definition

class ODEfunc(nn.Module):

    # init ODE variables
    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.norm1 = norm(dim)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = conv3x3(dim, dim)
        self.norm2 = norm(dim)
```

```
## Import the Adjoint Method (ODE Solver)
from torchdiffeq import odeint_adjoint as odeint

## Normal Residual Block Example

class ResBlock(nn.Module):

    #init a block - Convolve, pool, activate, repeat
    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(ResBlock, self).__init__()
        self.norm1 = norm(inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.norm2 = norm(planes)
        self.conv2 = conv3x3(planes, planes)

    #Forward pass - pass output of one layer to the input of the next
    def forward(self, x):
        shortcut = x
        out = self.relu(self.norm1(x))
        out = self.conv1(out)
        out = self.norm2(out)
        out = self.relu(out)
        out = self.conv2(out)

        return out + shortcut

## Ordinary Differential Equation Definition

class ODEfunc(nn.Module):

    # init ODE variables
    def __init__(self, dim):
        super(ODEfunc, self).__init__()
        self.norm1 = norm(dim)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = conv3x3(dim, dim)
        self.norm2 = norm(dim)
        self.conv2 = conv3x3(dim, dim)
        self.norm3 = norm(dim)
        self.nfe = 0

    # init ODE operations
    def forward(self, t, x):
      #nfe = number of function evaluations per timestep
        self.nfe += 1
        out = self.norm1(x)
        out = self.relu(out)
        out = self.conv1(out)
        out = self.norm2(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.norm3(out)
        return out


 ## ODE block
class ODEBlock(nn.Module):

    #initialized as an ODE Function
    #count the time
    def __init__(self, odefunc):
        super(ODEBlock, self).__init__()
        self.odefunc = odefunc
        self.integration_time = torch.tensor([0, 1]).float()
```

```
67:
68:     #foorward pass
69:     #input the ODE function and input data into the ODE Solver (adjoint method)
70:     # to compute a forward pass
71:     def forward(self, x):
72:         self.integration_time = self.integration_time.type_as(x)
73:         out = odeint(self.odefunc, x, self.integration_time, rtol=args.tol, atol=args.tol)
74:         return out[1]
75:
76:     @property
77:     def nfe(self):
78:         return self.odefunc.nfe
79:
80:     @nfe.setter
81:     def nfe(self, value):
82:         self.odefunc.nfe = value
83:
84: ## Main Method
85:
86: if __name__ == '__main__':
87:
88:
89:
90:     #Add Pooling
91:     downsampling_layers = [
92:          nn.Conv2d(1, 64, 3, 1),
93:          ResBlock(64, 64, stride=2, downsample=conv1x1(64, 64, 2)),
94:          ResBlock(64, 64, stride=2, downsample=conv1x1(64, 64, 2)),
95:      ]
96:
97:     # Initialize the network as 1 ODE Block
98:     feature_layers = [ODEBlock(ODEfunc(64))]
99:     # Fully connected Layer at the end
100:    fc_layers = [norm(64), nn.ReLU(inplace=True), nn.AdaptiveAvgPool2d((1, 1)), Flatten(), nn.Linea
101:
102:    #The Model consists of an ODE Block, pooling, and a fully connected block at the end
103:    model = nn.Sequential(*downsampling_layers, *feature_layers, *fc_layers).to(device)
104:
105:    #Declare Gradient Descent Optimizer
106:    optimizer = torch.optim.SGD(model.parameters(), lr=args.lr, momentum=0.9)
107:
108:    #Training Loop
109:    for itr in range(args.nepochs * batches_per_epoch):
110:
111:
112:        #init the optimizer
113:        optimizer.zero_grad()
114:
115:        #Generate training data
116:        x, y = data_gen()
117:        #Input Training data to model, get Prediction
118:        logits = model(x)
119:        #Compute Error using Prediction vs Actual Label
120:        loss = CrossEntropyLoss(logits, y)
121:
122:        #Backpropagate
123:        loss.backward()
124:        optimizer.step()
```

ODE solver (adjoint method)