Numpy.Dot: For 2-D arrays it is equivalent to matrix multiplication, and for 1-D arrays to inner product of vectors. For N dimensions it is a sum product over the last axis of a and the second-to-last of b. a = np.array([1,2,3]), a[::-1]=[3,2,1], Array Reverse.

```
>>> np.dot(3, 4)
12
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
>>> a = [1, 0], [0, 1]]
>>> b = [4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

Numpy.BroadCasting (*): Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Each dimension should satisfy: one of them is 1, or they are same. Broadcasting provides a convenient way of taking the outer product.

A:8 x 1 x 6 x 1 8x4x3 2 != 4
B: 7 x 1 x 5 2x1 both not 1
C:8 x 7 x 6 x 5 N/A

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \otimes \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_2b_1 & a_3b_1 \\ a_1b_2 & a_2b_2 & a_3b_2 \\ a_1b_3 & a_2b_3 & a_3b_3 \\ a_1b_4 & a_2b_4 & a_3b_4 \end{bmatrix}$$

Numpy.Matmul(Matrix Multiply): If both arguments are 2-D they are multiplied like conventional matrices. If one N-D, N>2. Shape(5,6,4,2)* Shape(5,6,2,4) = Shape(5,6,4,4). It is [0,0,:,:] * [0,0,:,:], [0,1,:,:] * [0,1,:,:]...

If one is 1-D, promote to 2-D and multiply. Can't multiply scalar, use *.

```
>>> a = [[1, 0], [0, 1]]
>>> b = [1, 2]
>>> np.matmul(a, b)
array([1, 2])
>>> np.matmul(b, a)
array([1, 2])
```

Array b: (2,)
First: (2,1)
Second: (1,2)

```
>>> a = np.arange(2*2*4).reshape((2,2,4))
>>> b = np.arange(2*2*4).reshape((2,4,2))
>>> np.matmul(a,b).shape
(2, 2, 2)
>>> np.matmul(a,b)[0,1,1]
98
>>> sum(a[0,1,:] * b[0,:,1])
98
```

Numpy.Reshape(Arg: Newshape): Gives a new shape to an array without changing its data.

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
```

Fortran way of array

```
>>> np.reshape(a, (3,-1))
s inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

-1 here means to calculate the size of last dimension

Back Propagation:

Input x

$$\begin{aligned} y &= g(x) \\ z &= h(x) \\ u &= f(y,z) \\ \ell &= u \end{aligned}$$

l is loss

```
u.grad = 1  #initializ
z.grad, y.grad, x.grad = 0
z.grad += u.grad * ∂f/∂z = ∂f/∂z
y.grad += u.grad * ∂f/∂y = ∂f/∂y
x.grad += y.grad * ∂g/∂x = ∂f/∂y ∂g/∂x
x.grad += z.grad * ∂h/∂x = ∂f/∂z ∂h/∂x
```

$$\text{x.grad} = \frac{\partial f}{\partial y}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial z}\frac{\partial h}{\partial x} = \frac{\partial \ell}{\partial x}.$$

$$v_{k+1} = f(v_0, \ldots, v_k)$$

$$\vdots$$

$$f\begin{cases} \forall i\ (v_0.\text{grad})[i] \mathrel{+}= \sum_j (v_{k+1}.\text{grad})[j]\ (\partial f(v_0,\ldots,v_k)[j]/\partial v_0[i]) \\ \vdots \\ \forall i\ (v_k.\text{grad})[i] \mathrel{+}= \sum_j (v_{k+1}.\text{grad})[j]\ (\partial f(v_0,\ldots,v_k)[j]/\partial v_k[i]) \end{cases}$$

Both Momentum, AdaGrad, RMSProp, Adam are discussing what is the best way of updating W after getting dx.

Momentum: Allow a velocity to build up along shallow directions. Mu usually 0.5,0.9,0.99.

```
#Momentum
v = mu * v - learningrate * dx #integrate velocity
x += v #Integrate position
```

Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

```
#Nesterov Momentum
v_prev = v
v = mu * v - learningrate * dx
x += -mu * v_prev + (1+mu) * v
```

Nest: Momentum + look ahead grad.

On the right it is the rewrote one.

RMSProp: Add element-wise scaling on grad based on history, also applied decay rate so learningrate not 0.

```
#RMSProp
Cache = decay_rate * cache + (1-decay_rate) * dx**2
x += -learningrate * dx / (np.sqrt(cache) + 1e-7)
```

Adam: Momentum + RMSProp. Accelerate + Scale + Decay

```
#Adam
m, v = 0
for t in ...
    dx = ...
    m = beta1 * m + (1-beta1)* dx #like momentum
    v = beta2 * v + (1-beta2)*(dx**2) #RMSProp
    mb = m/(1-beta1**t) #correct bias
    vb = v/(1-beta2**t) #correct bias
    x += -learningrate * mb / ( np.sqrt(vb) + 1e-7 ) #RMProp
```

在课件中，由于dx是向量，所以是取第i个元素出来计算

Sigmoid: (x)

Forward:

self.value = 1. / (1. + np.exp(-self.x.value))

Backward:

self.x.grad = self.x.grad + self.grad * self.value *(1 - self.value)

Mul: (x,y)

Forward:

self.value = self.x.value * self.y.value

Backward:

self.x.grad = self.x.grad + np.reshape(

np.sum(self.grad*self.y.value,axis=xred,keepdims=True), self.x.value.shape)

self.y.grad = self.y.grad + np.reshape(

np.sum(self.grad*self.x.value,axis=yred,keepdims=True),self.y.value.shape)

Tanh: (x)

Forward:

x_exp = np.exp(self.x.value)

x_neg_exp = np.exp(-self.x.value)

self.value = (x_exp - x_neg_exp)/(x_exp + x_neg_exp)

Backward:

self.x.grad = self.x.grad + self.grad * self.value *(1 - self.value)

SoftMax: (x)

Forward:

lmax = np.max(self.x.value,axis=-1,keepdims=True)

ex = np.exp(self.x.value - lmax)

self.value = ex / np.sum(ex,axis=-1,keepdims=True)

Backward:

self.x.grad = self.x.grad + self.grad * self.value *(1 - self.value)

L2 & L1 Regularization:

Theta conforms to some distribution $p(\Theta) \propto e^{-\frac{1}{2}\lambda||\Theta||^2}$

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}}\ \ell_{\text{train}}(\Theta) + \frac{1}{2}\lambda||\Theta||^2$$

$$\Theta \mathrel{-}= \eta\nabla_\Theta\, \ell_{\text{train}}(\Theta)$$

$$\Theta \mathrel{-}= \eta\lambda\Theta \qquad \text{(shrinkage)}$$

At equilibrum Θ_i is proportional to the "loss pressure" on Θ_i.

$$\Theta_i = \frac{-1}{\lambda}(\partial\ell/\partial\Theta_i)$$

第一行表示Theta满足右边的分布。第二行的推到：l* = -log(p(data|theta)) = -log(p(data)) – log(theta) = ltrain(theta) + 1/2xxx.第三，四行是对l*求导之后得dx,更新x就是这两步。第五行是说theta的最优值就是dx = 0,求出theta。

$$p(\Theta) \propto e^{-||\Theta||_1} \qquad ||\Theta||_1 = \sum_i |\Theta_i|$$

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}}\ \ell_{\text{train}}(\Theta) + \lambda||\Theta||_1$$

$$\Theta \mathrel{-}= \eta\nabla_\Theta\, \ell_{\text{train}}(\Theta)$$

$$\Theta_i \mathrel{-}= \eta\lambda\ \text{sign}(\Theta_i) \qquad \text{(shrinkage)}$$

At equilibrium (sparsity is difficult to achieve with SGD)

$$\begin{aligned} \Theta_i &= 0 \qquad \text{if } |\partial\ell/\partial\Theta_i| < \lambda \\ \partial\ell/\partial\Theta_i &= -\lambda \qquad \text{otherwise} \end{aligned}$$

Complex Step Directional Derivative:

$$f(x + i\cdot\varepsilon) = \frac{i^0\varepsilon^0}{0!}f(x) + \frac{i^1\varepsilon^1}{1!}f'(x) + \frac{i^2\varepsilon^2}{2!}f''(x) + \frac{i^3\varepsilon^3}{3!}f'''(x)$$

Take the first order one

$$f(x + i\epsilon) = f(x) + i(df/dx)\epsilon$$

左右两边相等。都表示成a+bi,而且f(x)和df/fx实数

$$\text{Im}(f(x + i\epsilon)) = \epsilon(df/dx)$$

If x is a tensor

$$[\nabla_x f(x)]\,\Delta x = \frac{\text{Im}(f(x + i\epsilon\Delta x))}{\epsilon}$$

Xavier Initialization: Initialize a weight matrix (or tensor) to preserve zero-mean unit variance distributions.

If we assume xi has unit mean and zero variance then we want to have zero mean and unit variance.

$$y_j = \sum_{j=0}^{N-1} x_i w_{i,j}$$

Xavier initialization randomly sets w(i,j) to be uniform in the following interval. N is number of neurons. Assuming independence this gives yj 0 mean and unit variance.

$$\left(-\sqrt{\frac{3}{N}}, \sqrt{\frac{3}{N}}\right)$$

Implementation: np.prod here calculates the mul of all elements.

```
def xavier(shape):
    sq = np.sqrt(3.0/np.prod(shape[:-1]))
    return np.random.uniform(-sq,sq,shape)
```

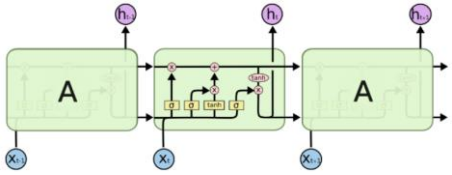
The above is reasonable for tanh and assumes that we sum over all but the last index. Reason it is good, see standford chapt6. Good for symmetric activation function.
For example, an image convolution filter has shape (H, W, C1, C2) and we sum over the first three indices. A single filter with shape H,W,C1 produce one number each time

Kaiming Initialization: A ReLU nonlinearity reduces the variance.

So we use the following one:

$$\left(-\sqrt{\frac{6}{N}}, \sqrt{\frac{6}{N}}\right)$$

LSTM:



The highway path goes across the top and is called the Carry:
The highway path C (across the top of the figure) is gated. The product F * C is componentwise (as in Numpy). We say F gates C. F is called the "forget gate".
ResNet is to PlainNet (Resnet connects what plain net is not connected) what LSTM is to RNN (LSTM adds a highway), kind of.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$
$$h_t^l = o \odot \tanh(c_t^l)$$

Gates: 1. what information we're going to throw away from the cell state. 2. what new information we're going to store in the cell state. 3. Filter output 4. Filter cell state

Batch Normalization: For each value in a tensor, batch normalization computed the mean and variance over the batch and renormalizes the value to have unit variance and zero mean.
It is typically used just prior to a nonlinear activation function.
Pros: Improve grad flow, higher learning rate, allow adjust y's range

$$\text{Batchnorm}(x) = \frac{x - \hat{\mu}}{\hat{\sigma}}$$

$$\hat{\mu} = \frac{1}{B}\sum_b x_b$$

$$\hat{\sigma} = \sqrt{\frac{1}{B-1}\sum_b (x_b - \hat{\mu})^2}$$

At training time backpropagation goes through the calculation of $\hat{\mu}$ and $\hat{\sigma}$.

At test time $\hat{\mu}$ and $\hat{\sigma}$ are fixed rather than computed from the batch.

Batch Norm with SGD: Consider the following.

$$y.\text{value}[\cdots] \mathrel{+}= w.\text{value}[\cdots]\, x.\text{value}[\ldots]$$

$$w.\text{grad}[\cdots] \mathrel{+}= y.\text{grad}[\cdots]\, x.\text{value}[\cdots]$$

Replacing x by $x/\hat{\sigma}$ seems related to RMSProp for the update of w.

Karpathy Conjecture: Consider the following.

A normalization layer $y = \alpha(x + \beta)$ can be used in place of batch normalization as long β is initialized to $-\hat{\mu}$ and α is initialized to $1/\hat{\sigma}$.

Here $\hat{\mu}$ and $\hat{\sigma}$ should be computed only after earlier normalizations have been properly initialized.

Dropout: randomly set some neurons to zero in the forward pass. Used before relu. Kind of training a subset of NN

$$\begin{cases} \mu_i = 0 \text{ with probability } \alpha \\ \mu_i = 1 \text{ with probability } 1-\alpha \end{cases} \qquad y_i = \text{Relu}\left(\sum_j W_{i,j}\, \mu_j x_j\right)$$

At train time we have

$$y_i = \text{Relu}\left(\sum_j W_{i,j}\, \mu_j x_j\right)$$

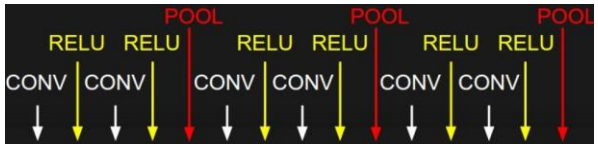
At test time we have

$$y_i = \text{Relu}\left((1-\alpha)\sum_j W_{i,j}\, x_j\right)$$

At test time we use the "average network".

during test: **a = w0*x + w1*y**
during train:
E[a] = ¼ * (w0*0 + w1*0
w0*0 + w1*y
w0*x + w1*0
w0*x + w1*y)
= ¼ * (2 w0*x + 2 w1*y)
= ½ * **(w0*x + w1*y)**

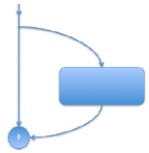
CNN: Each layer a bunch of filters, each filter produce a plane of numbers. All the numbers in one layer forms a 3-D matrix. (2-D plane for 1 filter, n filters)



Pure Highway: The diversion Di(Li) fits the residual of the identity function

$$L_{i+1} = L_i + D_i(L_i)$$

Deep Residual Network: Resnets have a "highway path" connecting the input to the output. This preserves gradients.

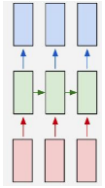


Recurrent Neural Network: We can process a sequence of vectors x by applying a recurrence formula at every time step

$$h_t = f_W(h_{t-1}, x_t)$$

new state — old state — input vector at some time step

some function with parameters W



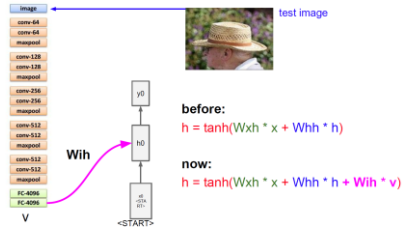
Notice: the same function and the same set of parameters are used at every time step.

(Vanilla) Recurrent Neural Network

$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t)$$

$$y_t = W_{hy} h_t$$

Combination of RNN to CNN



GRU:The highway path is H.

$$H_{t+1} = (1 - I_t) * H_t + I_t * D_t$$
$$I_t = \sigma(W_I[X_t, H_t])$$
$$D_t = \tanh(W_D[X_t, \sigma(W_r[x_t, H_t]) * H_t])$$