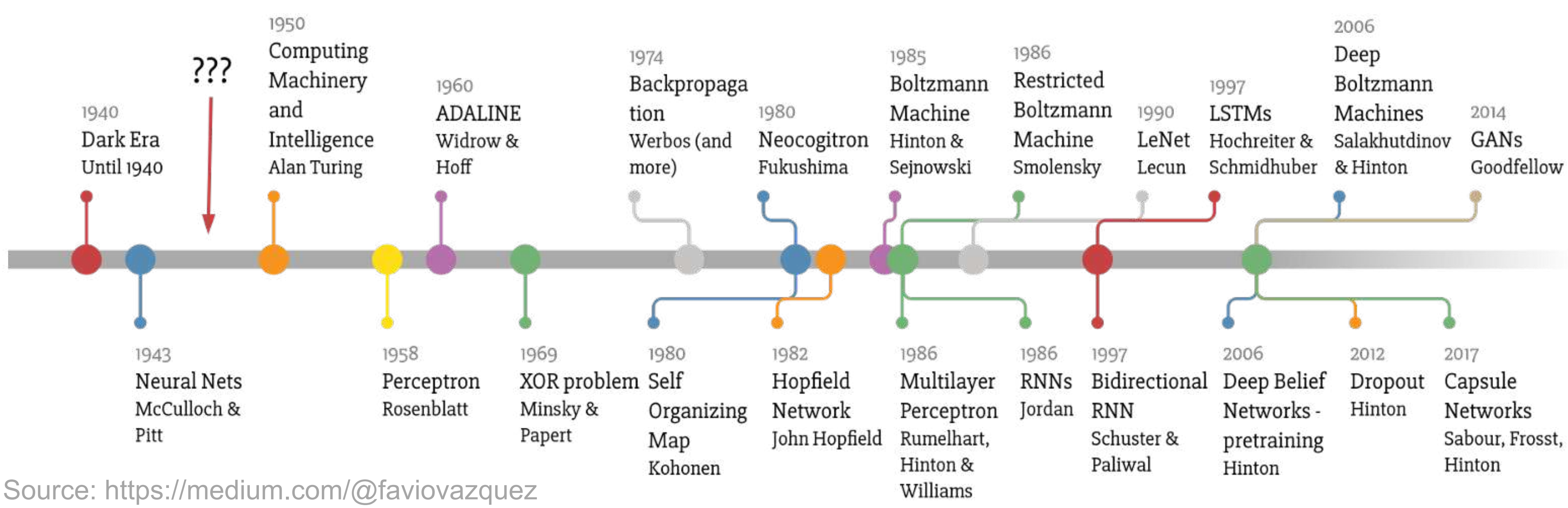
NUS-ISS
Pattern Recognition using Machine Learning System

The long march to the deepest network

by Dr. Tan Jen Hong

The deeper the tougher
Breaking the winter

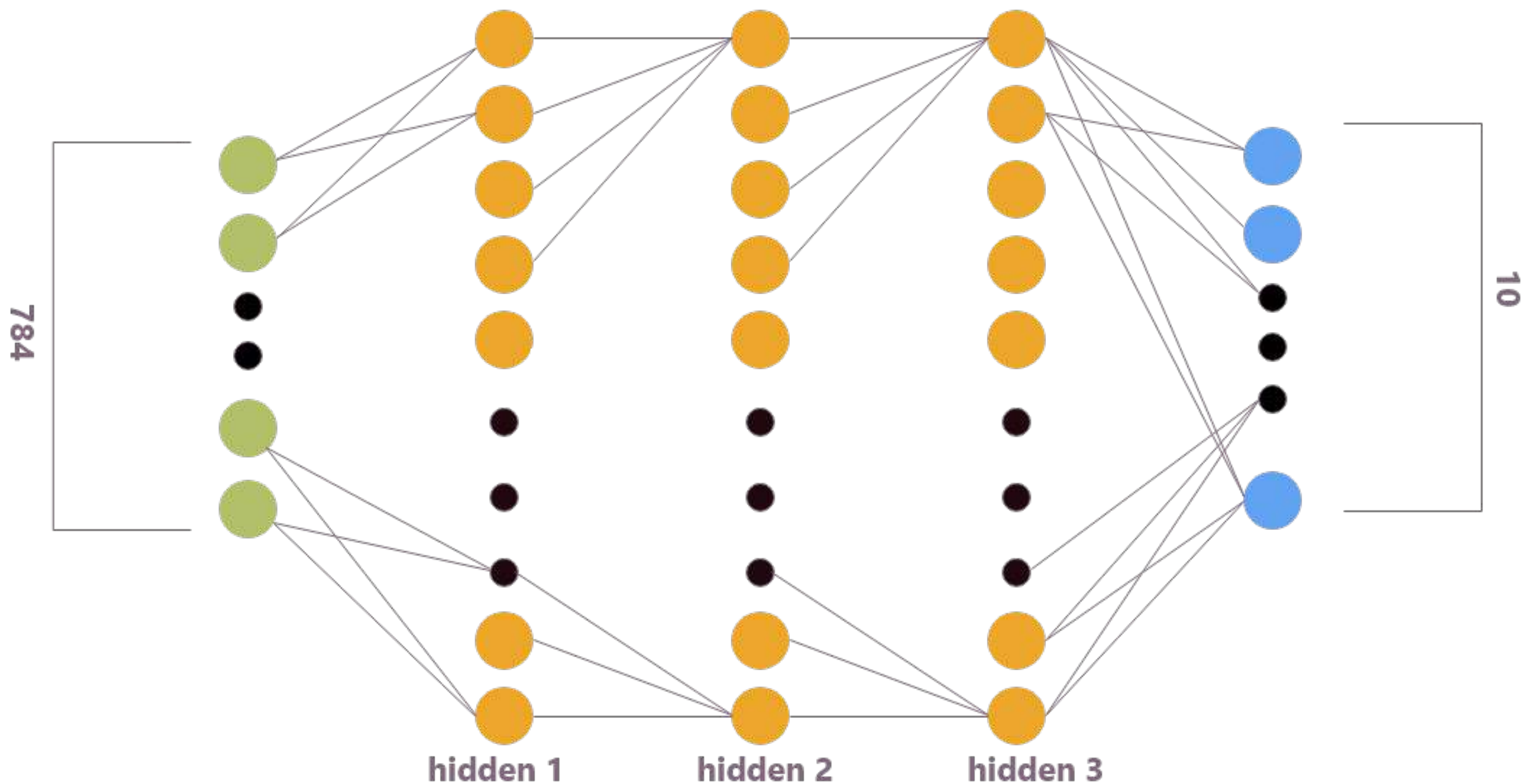
- The deeper a net, the better it performs
- But deeper the net, the harder to train; sometimes not possible to train
- Why?
- Why deep learning came so late?



Source: https://medium.com/@faviovazquez

Getting smaller and smaller ....

- Look at the gradient...



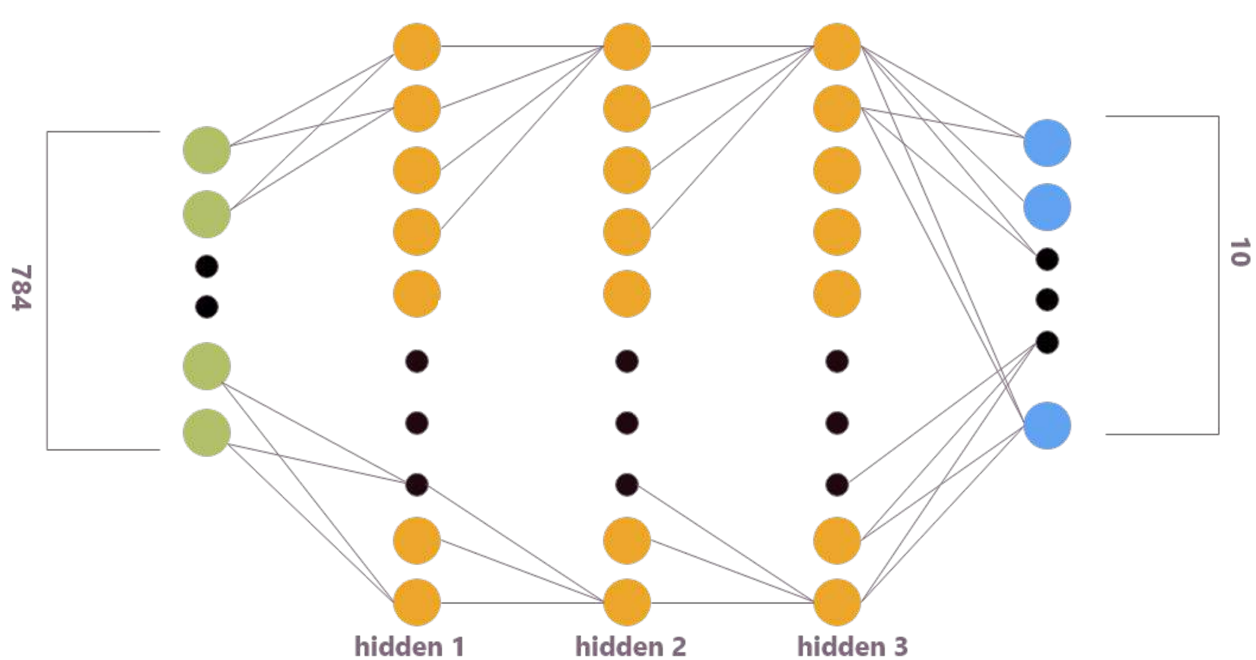
Source: By Manik Soni

Getting smaller and
smaller ....

- Backpropagation: For each neuron, calculate the gradient of loss (error) with respect to weight

- The gradient gets smaller and smaller when it moves backward in the net

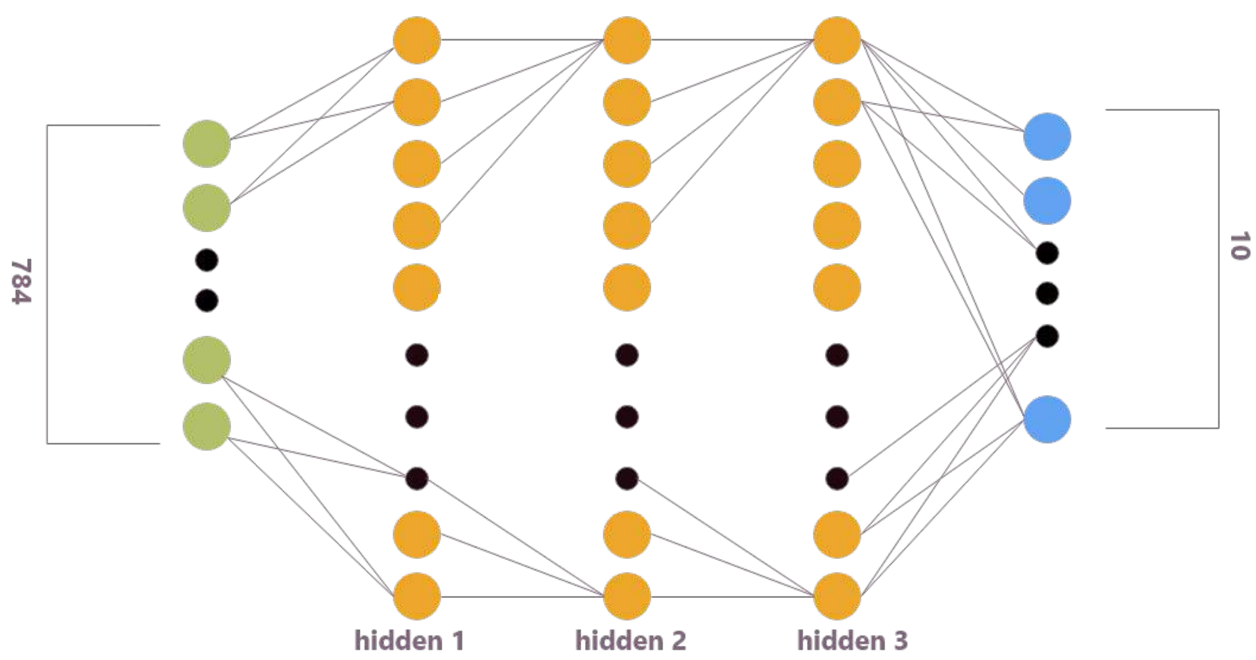
- Consequences: the earlier layers learn very slowly



Source: By Manik Soni

The importance
of the earlier layers

- Earlier layers are the key to good performance
- Earlier layers can detect simple patterns underlying input
- Earlier layers are feature extractors
- If feature extractors fail, garbage in garbage out for the classifier



Source: By Manik Soni

How to get the earlier layers trained?

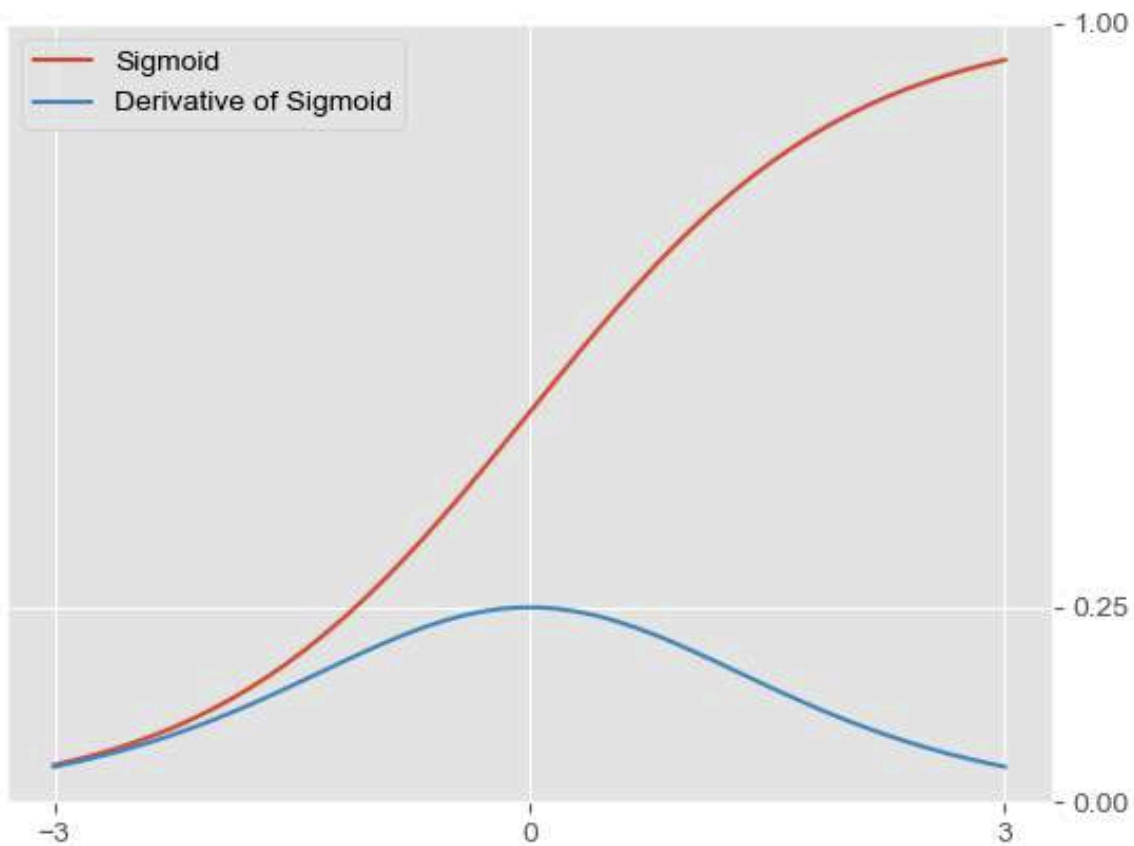
3 main strategies to start

- First, pick the right activation function
- Second, give the weights proper values to start
- Third, restrict the weights

First, pick the right activation function

The problem with Sigmoid

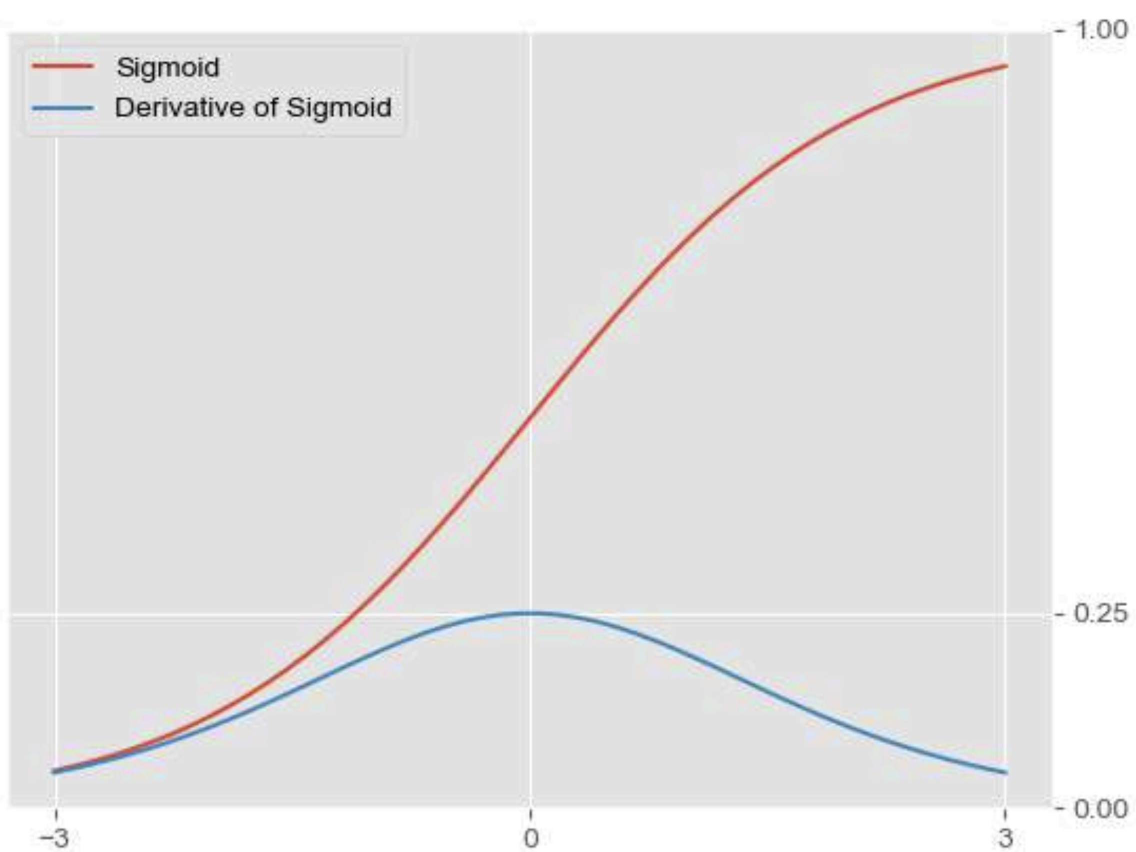
Small gradient



- When sigmoid function value either too high or two low, its derivative very small << 1
- Gradient vanish and poor learning as a consequence
- Occur when weights are pooly initialized (with large negative or positive values)

The problem with Sigmoid

Small gradient



- Even if weights initialized nicely, the largest derivative value is still around 0.25
- In the path of backpropagation, the multiplication of this small number leads to vanishing gradient
- Example: 0.25 x 0.25 x 0.25 x 0.25 gives 0.004, very small!
- Consequence: hard to update earlier layers, many static or dead neurons

Better derivative
from a better function

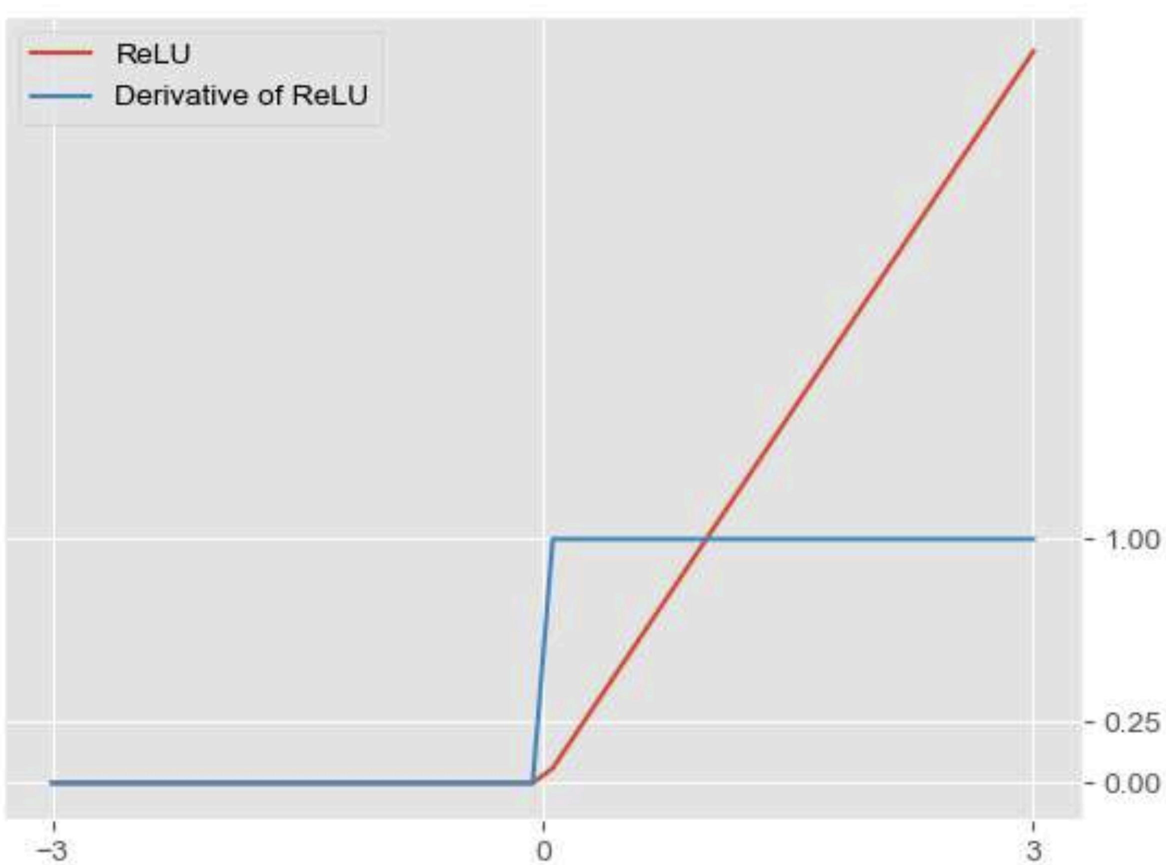
- Thus rectified linear unit is used

$$f(x) = \max(0, x)$$

- Derivative value is 1 when x > 0

- So: 1 x 1 x 1 x 1 gives 1. No matter how many times you multiply, always 1

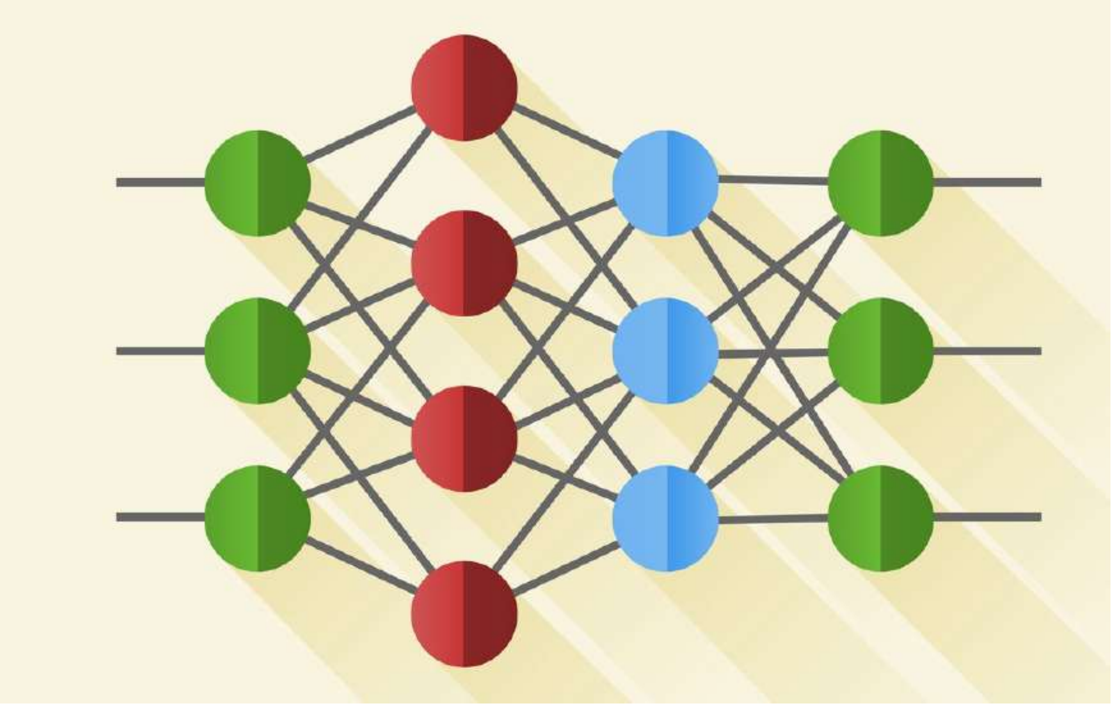
- Consequence: Good for update and prevent gradient vanish



Second, give the weights proper values

Initialization

Something to start with

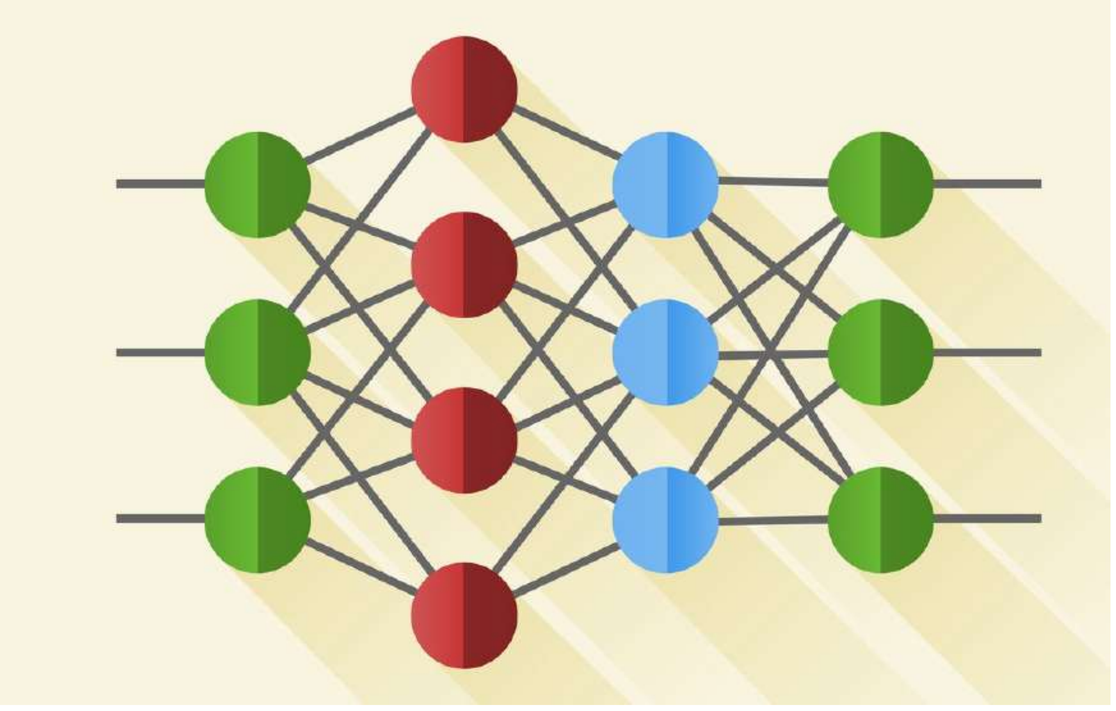


Source: https://towardsdatascience.com/nns-aynk-c34efe37f15a

- Need to give weights some values to start for the training
- Can't set all to 0, nothing will be learned in this case
- If weights are too small, signal shrinks as it passes through each layer; at later stage, neurons are 'dead', almost no activation coming out from neurons
- If weights are too large, signal grows massive as it passes through each layer; at later stage, neurons are 'saturated', network becomes unstable

Xavier initialization

proposed in 2010



Source: https://towardsdatascience.com/nns-aynk-c34efe37f15a

- At that time designed for Sigmoid and tanh function
- Initialize biases to be 0, the weight of each layer is

$$w \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

- n is the number of inputs to a neuron in the layer
- U is uniform distribution with a specified interval
- The weight is drawn randomly from the distribution

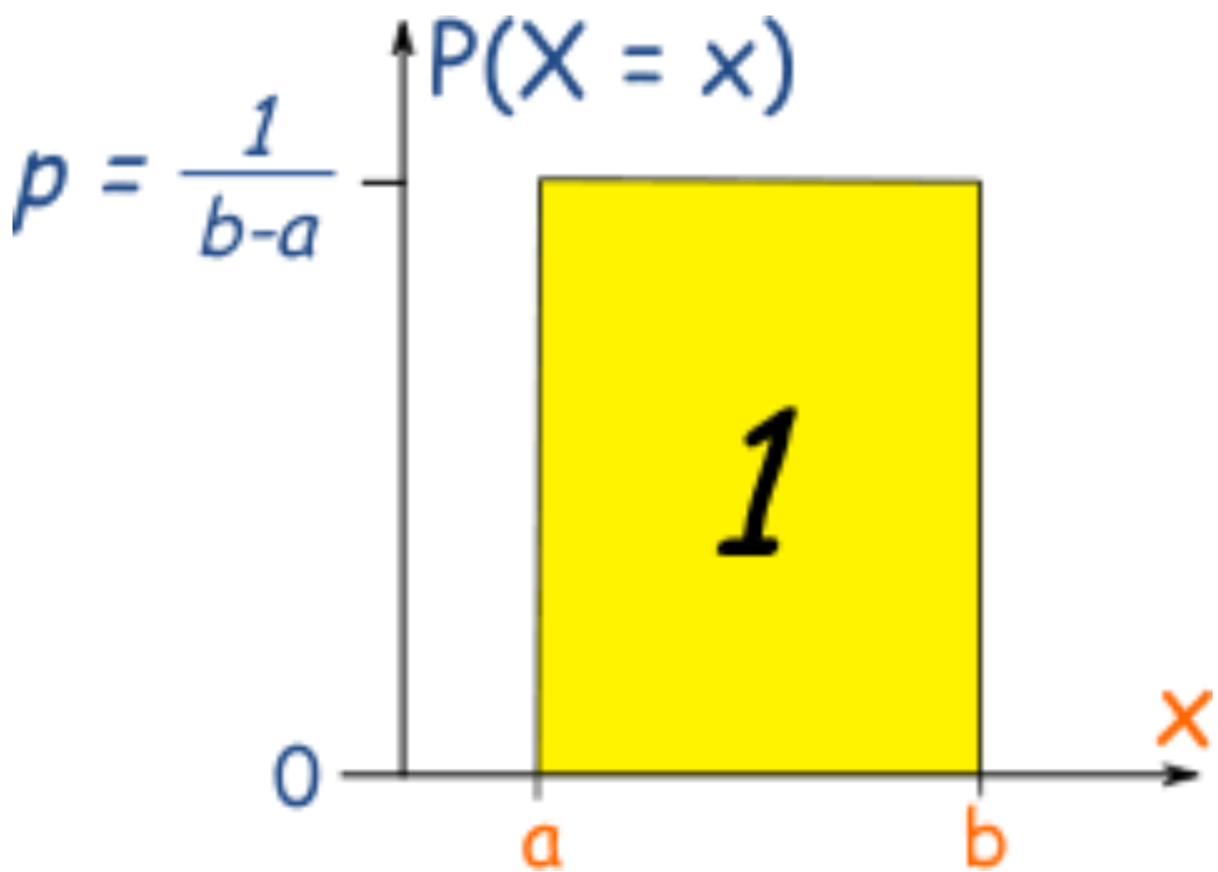
Xavier initialization

Uniform distribution

- Uniform distribution: a distribution that has a constant probability

- In the case of Xavier initialization:

$$a = -\frac{1}{\sqrt{n}} \qquad b = \frac{1}{\sqrt{n}}$$



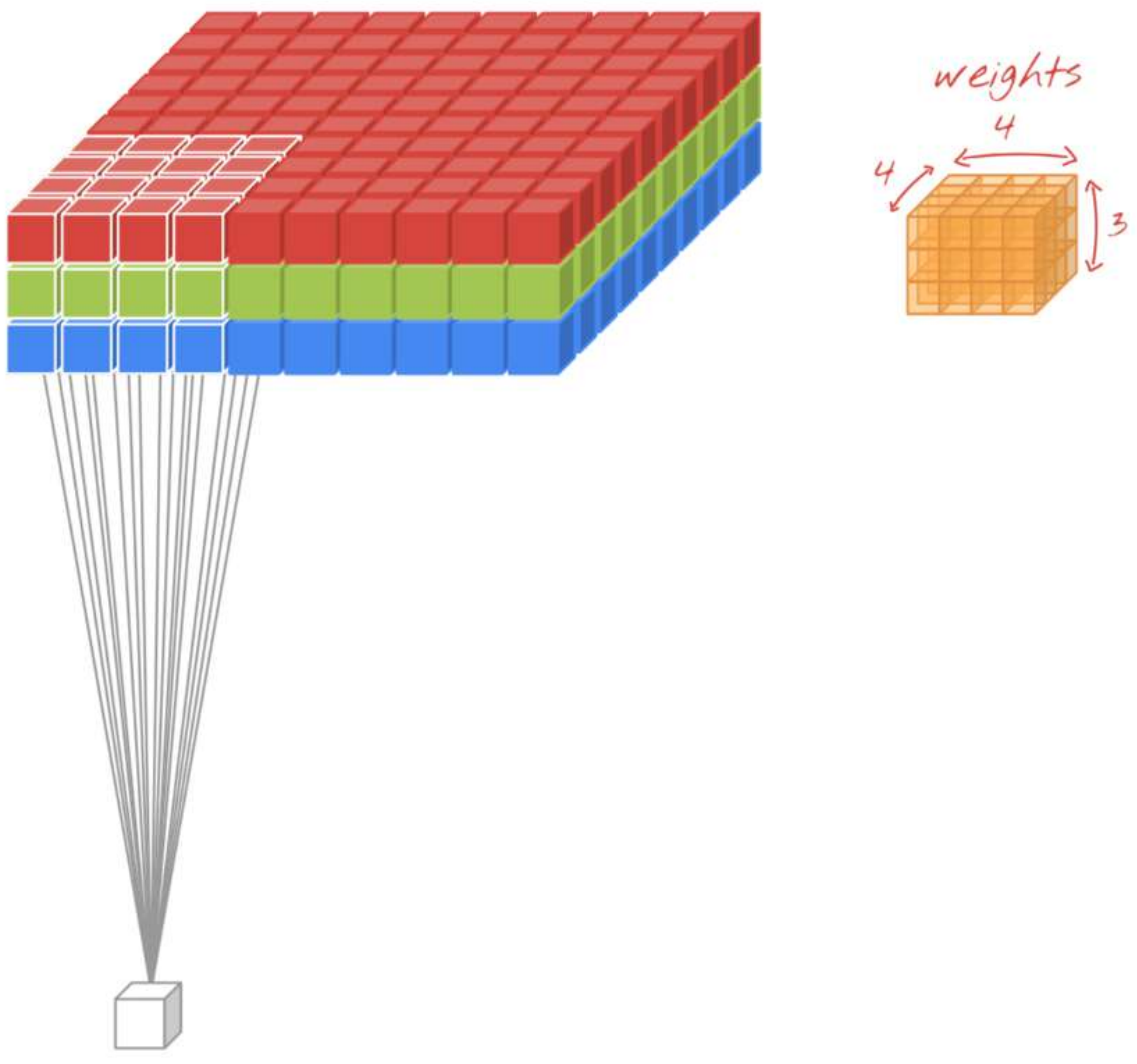
Source: https://www.mathsisfun.com/data/random-variables-continuous.html

The number of inputs?

To a neuron

- In literature the number of inputs to a neuron in a layer is called fan-in

- The number of inputs?



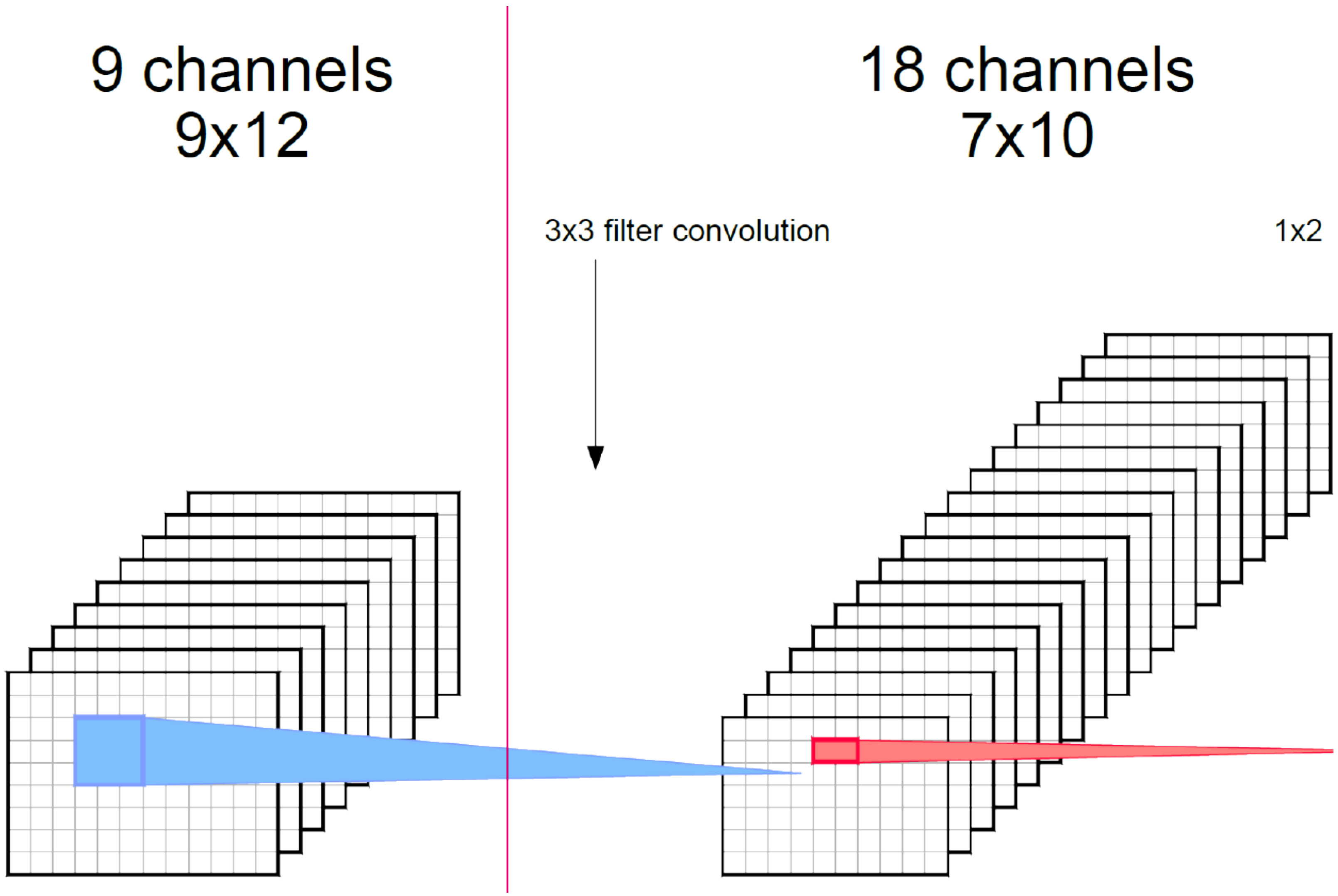
Source: https://twitter.com/martin_gorner

The number of inputs?

To a neuron

- What is the number of inputs to a neuron in the layer that has 18 channels?

- (3 x 3) x 9



Source: https://towardsdatascience.com/nns-aynk-c34efe37f15a

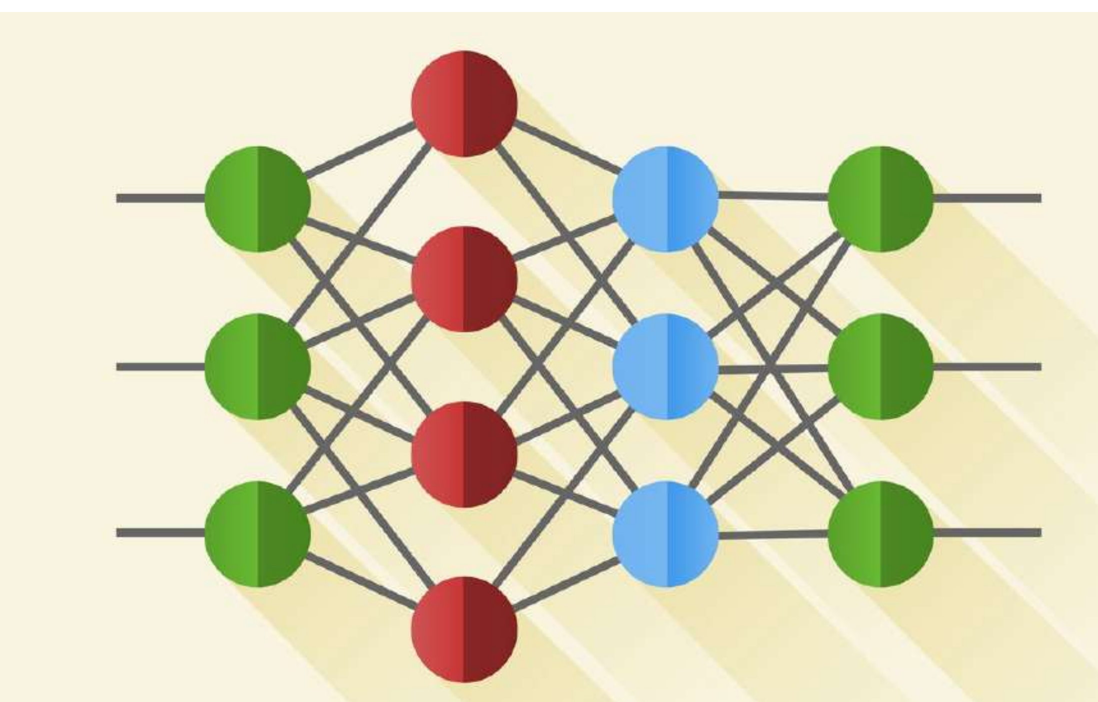
He initialization

proposed in 2015

- Designed for ReLU
- Initialize biases to be 0, the weight of each layer is

$$w \sim N\left(0, \frac{2}{n}\right)$$

- n is the number of inputs to a neuron in the layer
- N is a normal distribution with a zero mean and a variance of 2/n

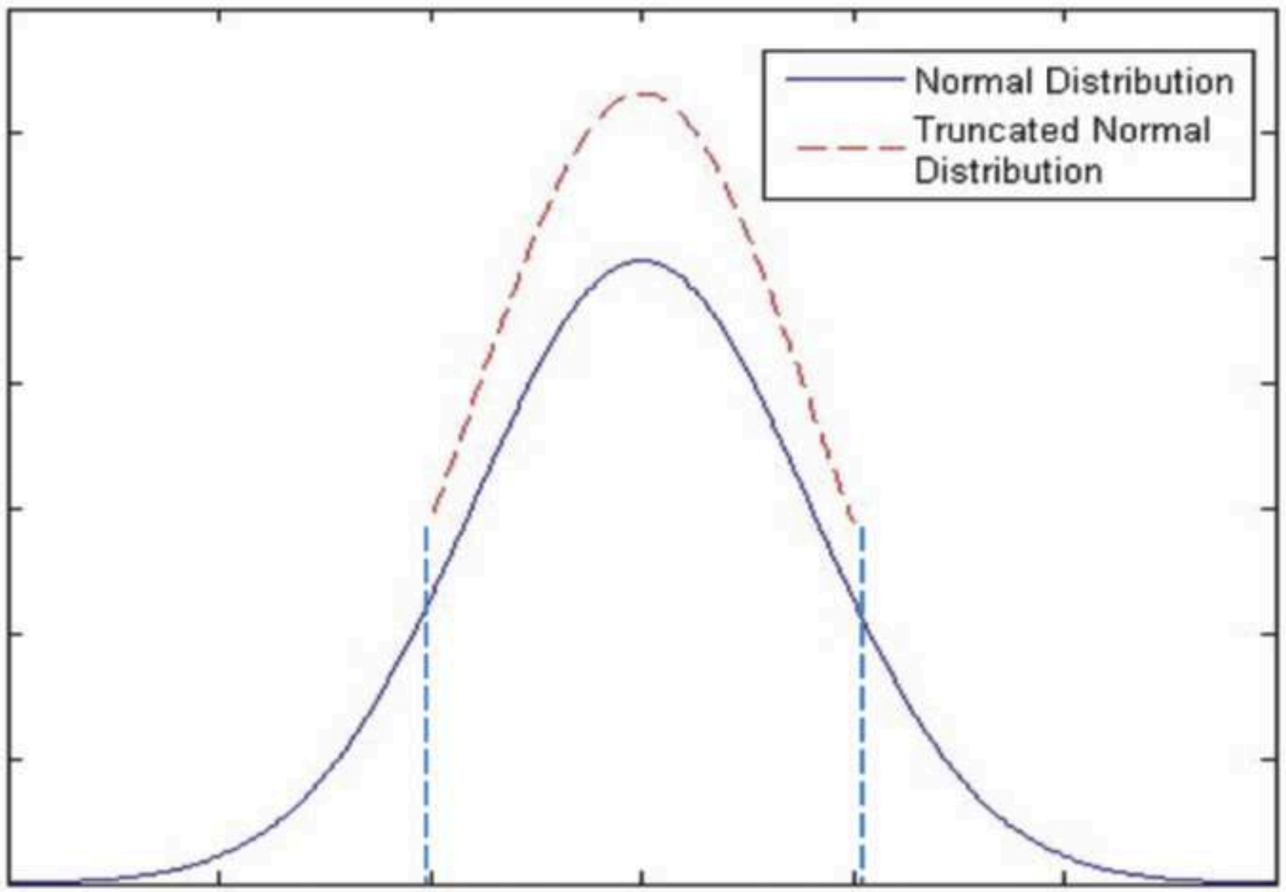


Source: https://towardsdatascience.com/nns-aynk-c34efe37f15a

He initialization

Truncated normal distribution

- In actual application, the normal distribution is truncated to avoid unnecessary large values



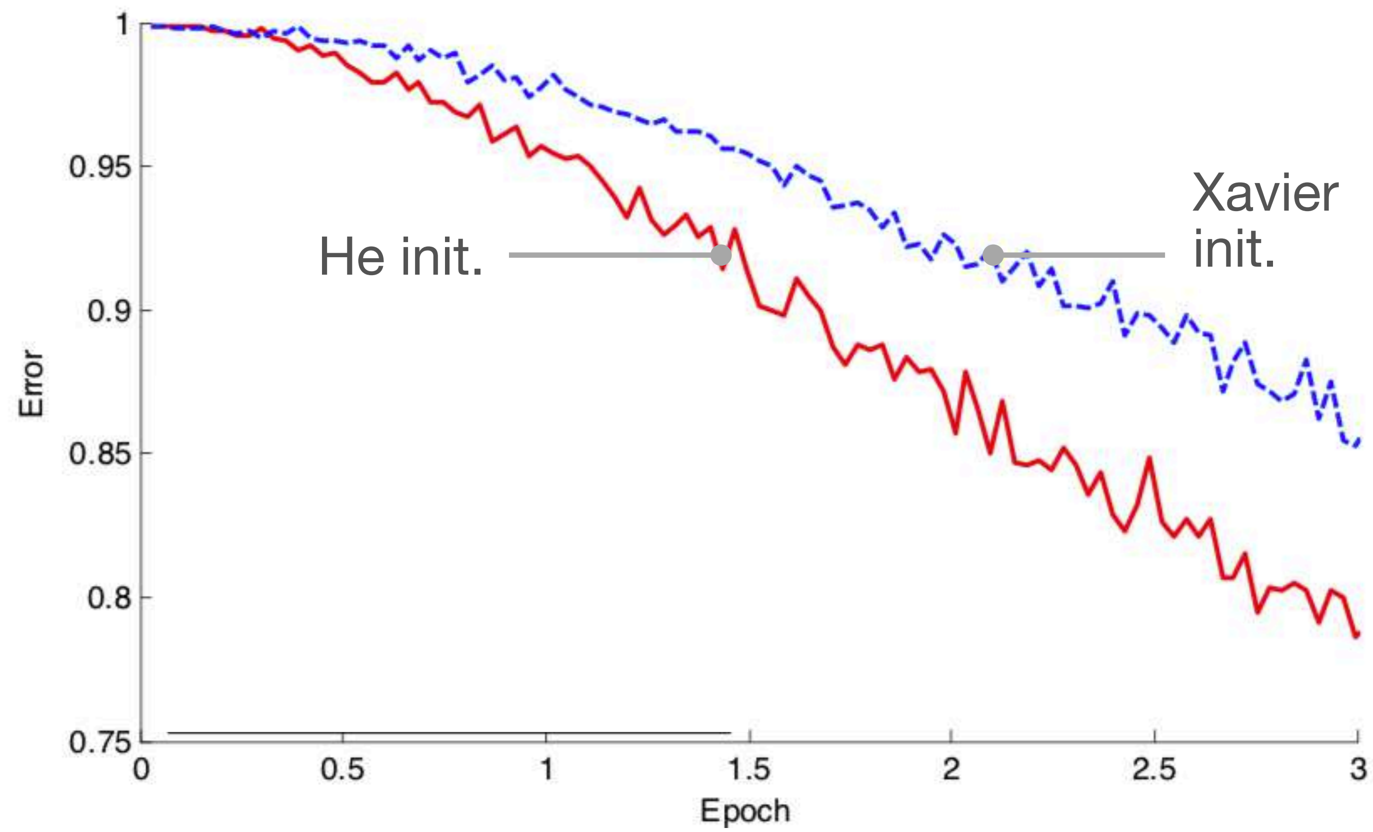
$$-\sqrt{\frac{2}{n}} \qquad \sqrt{\frac{2}{n}}$$

Source: doi: 10.3141/2315-07

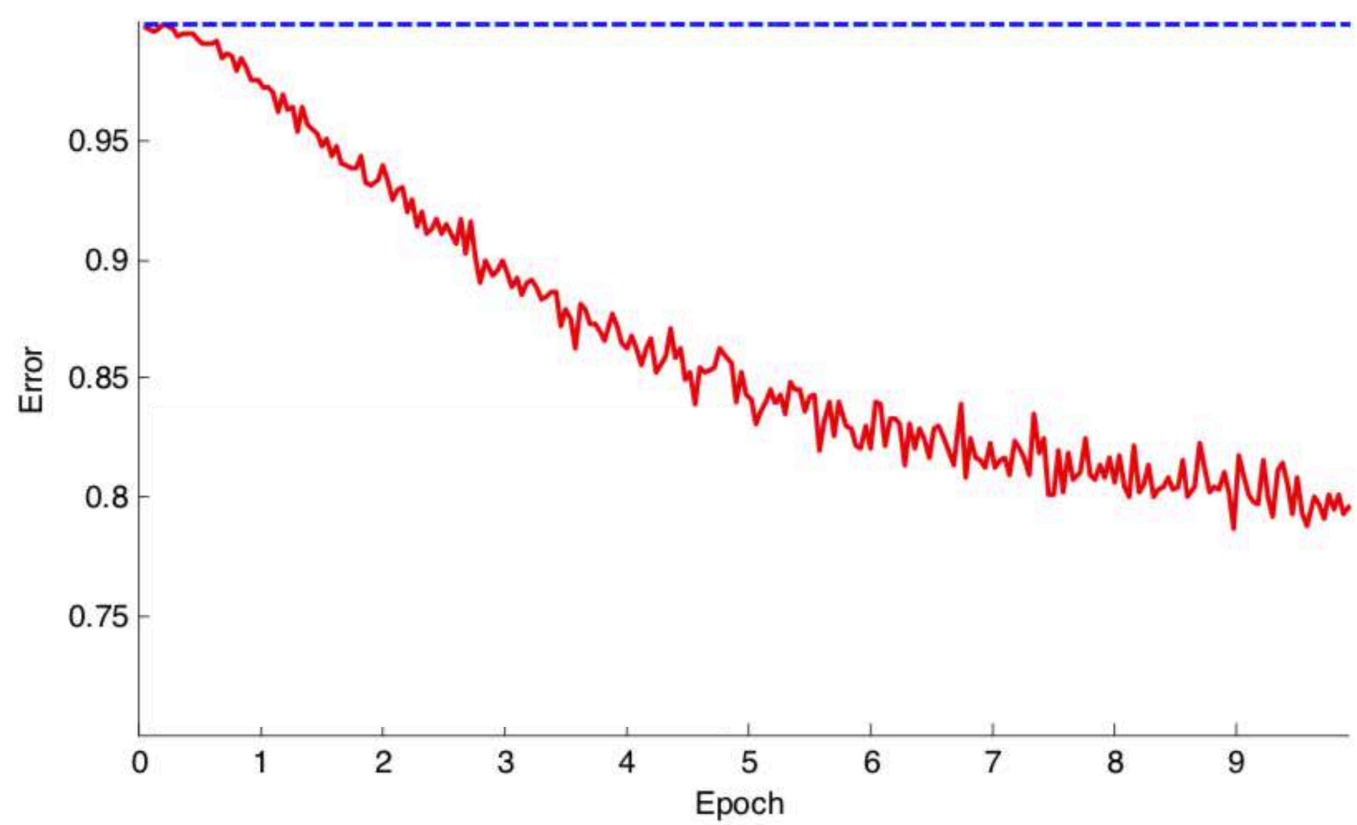
He initialization

proposed in 2015

22-layer
large model



30-layer
small model



Source: https://arxiv.org/pdf/1502.01852.pdf

Default in Keras
Initialization

- Keras by default uses Xavier initialization for weights, and zero initialization for bias

- See the code to make the change

- Note: many times it really needs to try to see which setup works better; no hard rules!

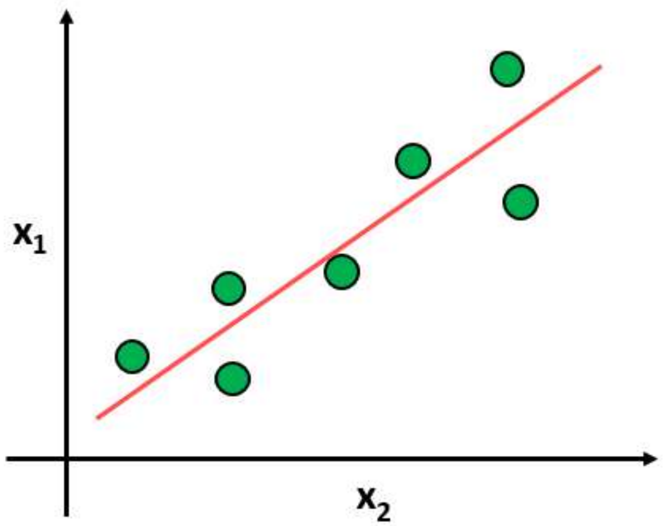
```
> from tensorflow.keras.initializers import he_normal

> x    = Dense(256,activation='relu',kernel_initializer=he_normal(33))(x)
```

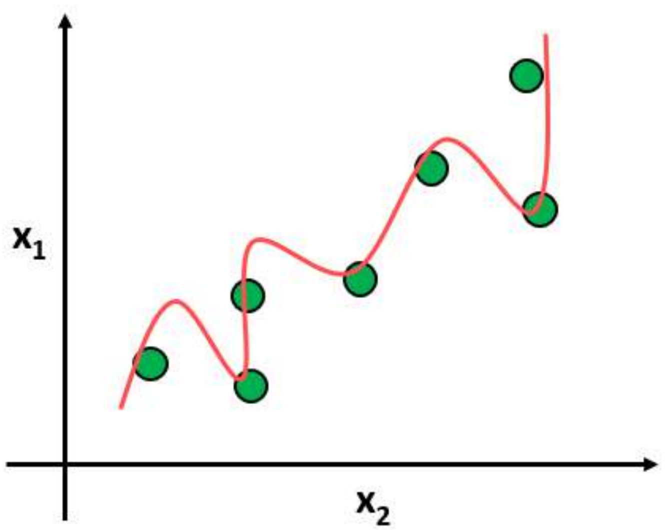
seed number

Third, restrict the weights

The problem with large values

The unstability



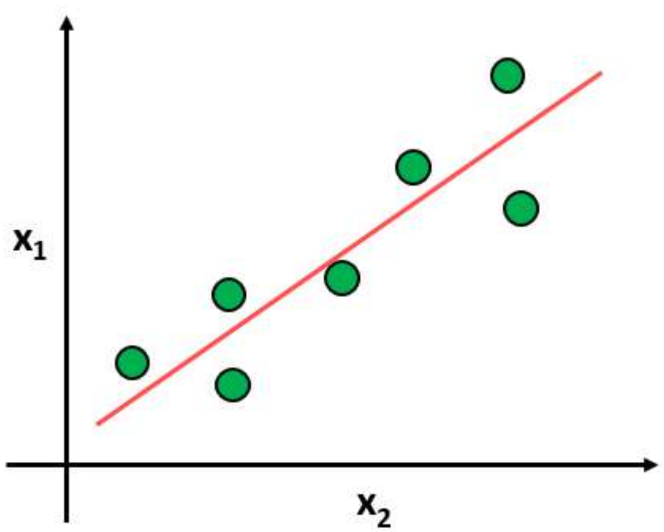


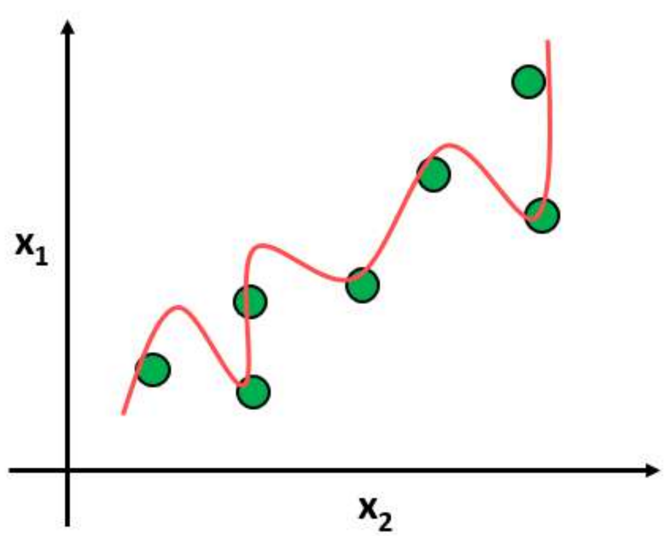
Source: https://www.globalsoftwaresupport.com/regularization-machine-learning/

- The longer we train, the weights will become more adapted to training data
- Eventually, weights will grow in order to handle specifics example in training data
- But, large weights make net unstable. Minor variation in values or addition of noise will result in big differences in output
- Prefer simpler model with smaller weights

The problem with large values

Regularization





Source: https://www.globalsoftwaresupport.com/regularization-machine-learning/

- A need to control the magnitude of the weight. This is called regularization
- Add a penalty to loss function; the penalty should be proportional to the magnitude of the weight
- L2 regularization is more often used; it calculates the sum of squared values of the weights
- L2 approach also called 'weight decay' in the field of neural networks, or 'shrinkage' in statistics

L2 regularization

The math

- Let J be the loss function, J_r the loss function with regularization, w the weight, η the learning rate, and α the parameter to control L2 regularization

The updated loss function

$$J_r = \frac{\alpha}{2}w^2 + J$$

The derivative

$$\frac{\partial J_r}{\partial w} = \alpha w + \frac{\partial J}{\partial w}$$

The algebraic expression to update a weight

$$w \leftarrow w - \eta \frac{\partial J_r}{\partial w}$$

Re-arrange the expression

$$w \leftarrow \underbrace{(1 - \alpha\eta)}_{\text{weight decay}}w - \eta \frac{\partial J}{\partial w}$$

How about L1?

Less used

- L1 calculates the sum of the absolute values of weights
- No clean algebraic solution and can be computational inefficient
- Encourage weights to be 0.0 if possible. Consequence: sparse weights (weights with more zero values)

$$J_r = \alpha \left| w \right| + J$$

Used in Keras

keras.regularizers

- In Keras it is easy to setup regularization
- Note: the penalty is applied on per-layer basis. Not all type of layer support regularization, and not all support in the same way
- Dense, Conv1D, Conv2D and Conv3D have a unified API

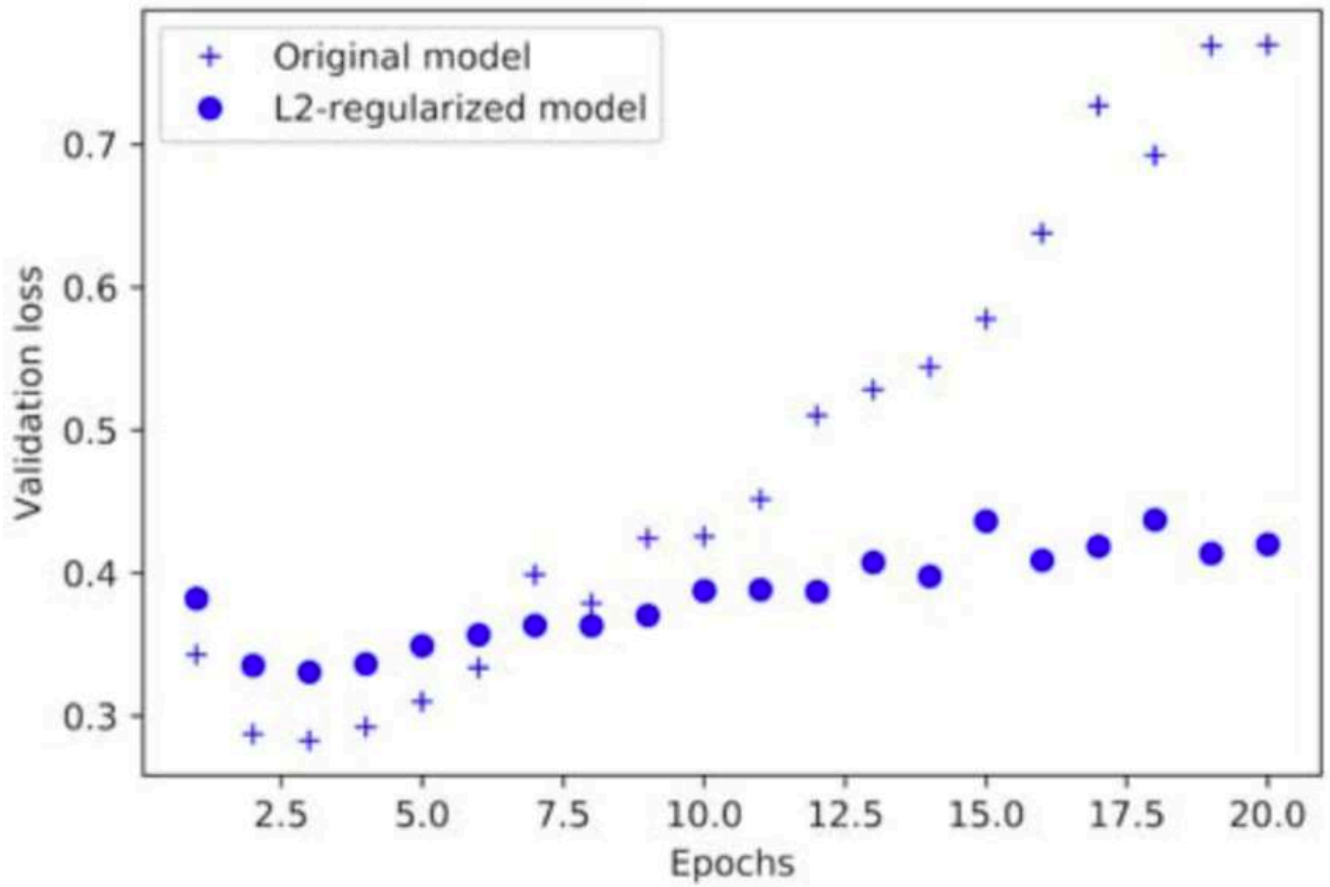
```
> from tensorflow.keras import regularizers

> x    = Dense(256,activation='relu',kernel_regularizer=regularizers.l2(0.01)(x)
```

Must a value between 0 (no penalty) to 1 (full penalty)

Used in Keras

keras.regularizers

```
> from tensorflow.keras import regularizers

> model = models.Sequential()
> model.add(Dense(16,
                  kernel_regularizer=regularizers.l2(0.001),
                  activation='relu',
                  input_shape=(10000,)))
> model.add(Dense(16,
                  kernel_regularizer=regularizers.l2(0.001),
                  ctivation='relu'))
> model.add(Dense(1, activation='sigmoid'))
```



Source: 'Deep learning with python' by Francois Chollet

But, even with these, we still have problems ....

Overfitting

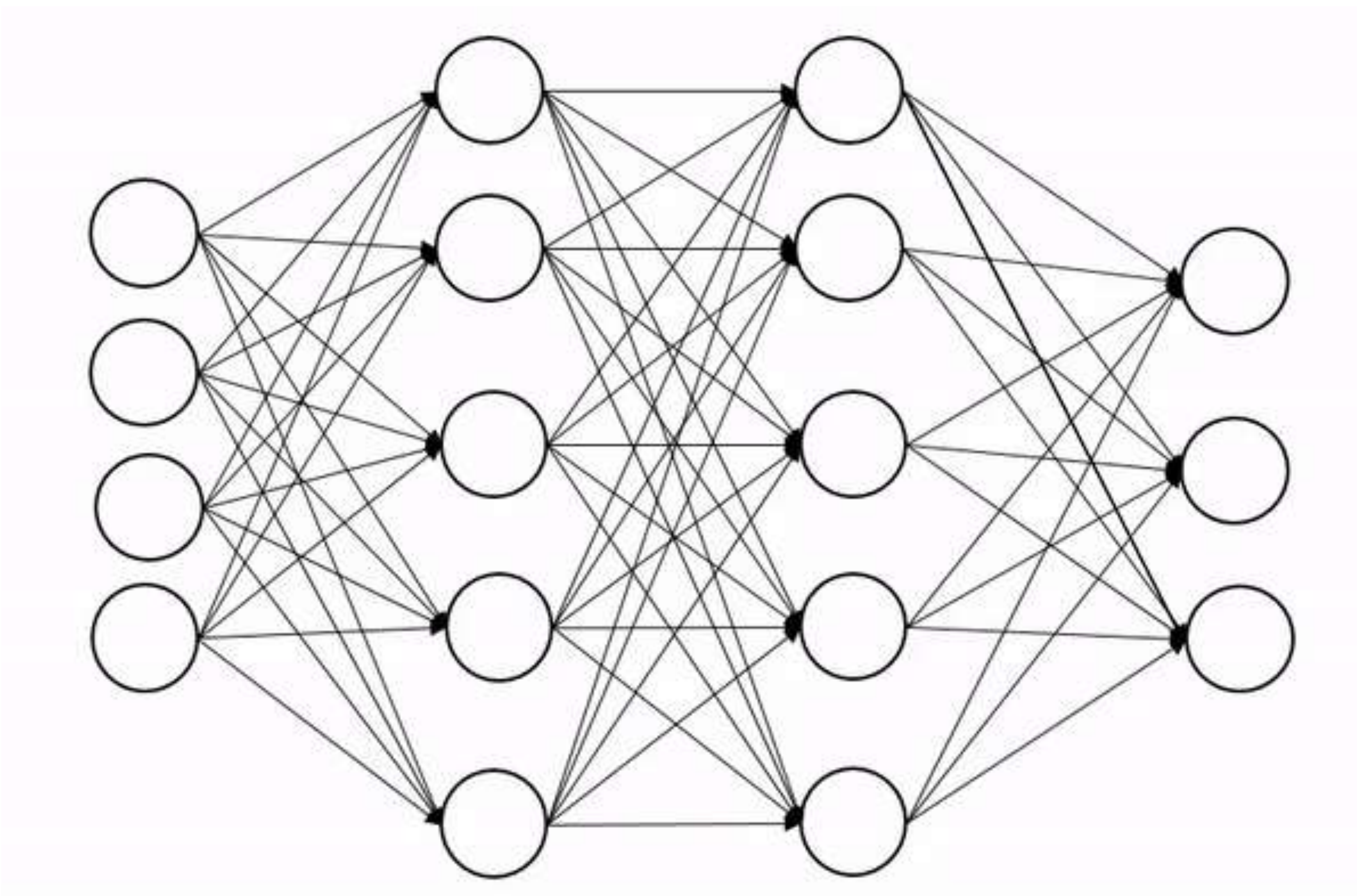
The pain, always

- Deep, large neural nets on small dataset: most of the time, they learn more on statistical noise in the data rather than key features
- Consequence: when new data come in, error increases; poor generalization
- Possible solutions: get many many neural network trained on same dataset. To perform classification, do average on prediction from each model (ensemble)
- However, not feasible in practice ....

Dropout

proposed in 2014

- Randomly dropping out neurons in the training phase. The 'dropping out' can be performed simply by setting the activation output from a neuron to zero



Source: https://www.globalsoftwaresupport.com/regularization-machine-learning/

Dropout
proposed in 2014

Source: https://torontolife.com/tech/ai-superstars-google-facebook-apple-studied-guy/

- The idea came out by Geoffrey Hinton, inspired by a fraud-prevention mechanism used by banks

- In his own words,

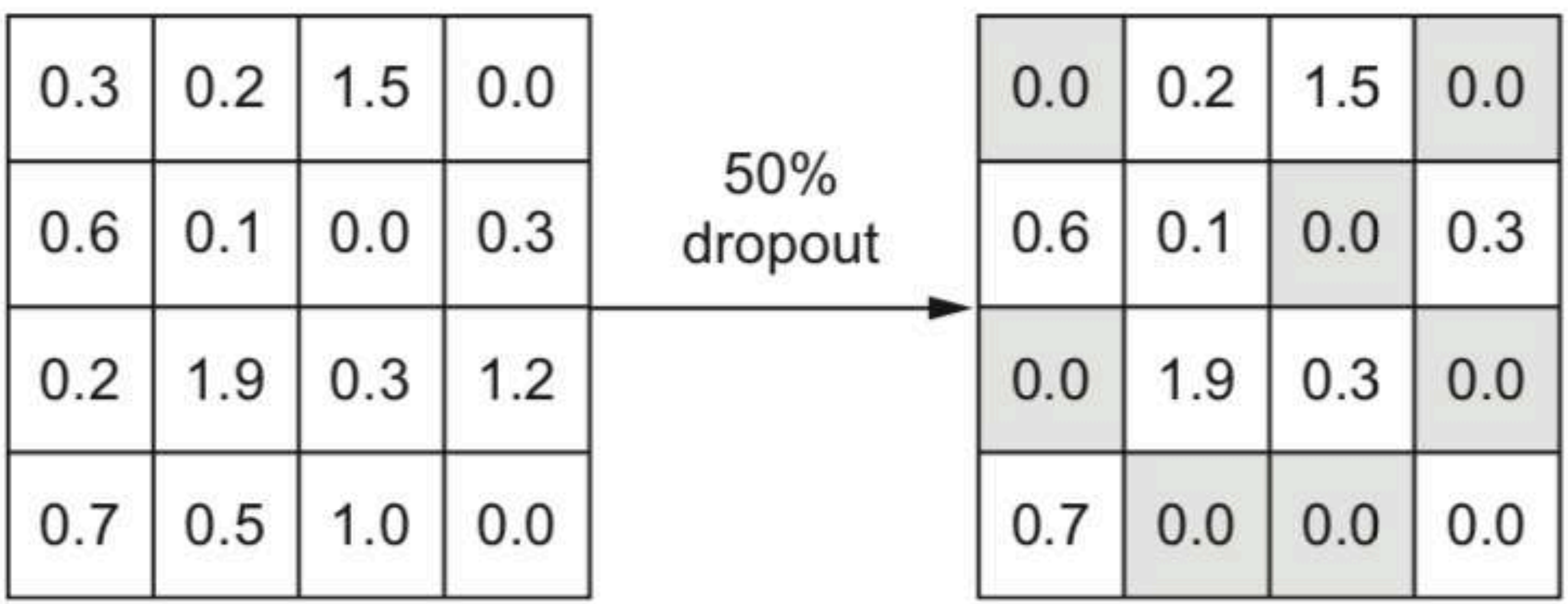
*I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. **This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.***

Dropout

In the form of tensor

- Note: the zeros will be applied to different neurons in different training epoch

- But the zeros are applied only during training phase, not in testing / validation

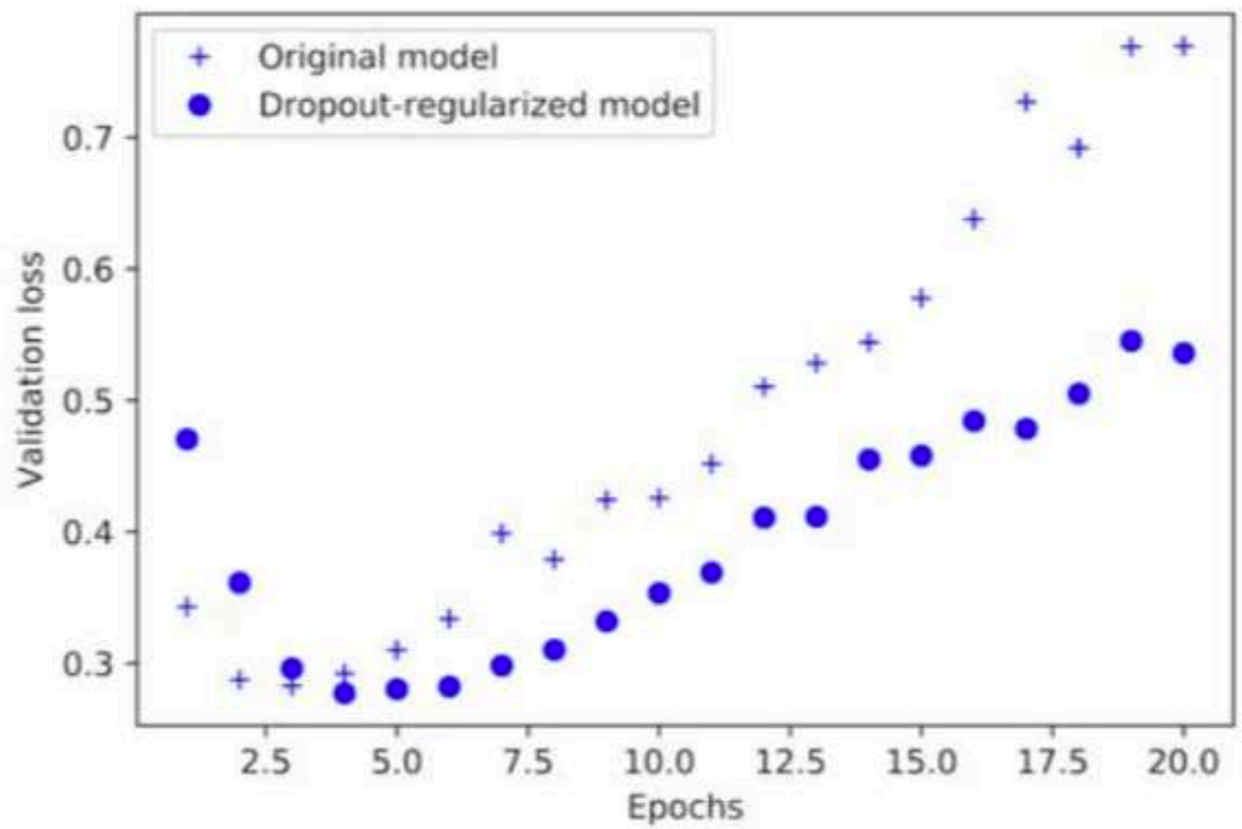


Source: 'Deep learning with python' by Francois Chollet

Dropout in Keras

comparison

```
> from tensorflow.keras.layers import Dropout

> model = models.Sequential()
> model.add(Dense(16,
                  activation='relu',
                  input_shape=(10000,)))
> model.add(Dropout(0.5)
> model.add(lDense(16,
                   activation='relu'))
> model.add(Dropout(0.5)
> model.add(Dense(1, activation='sigmoid'))
```



Source: 'Deep learning with python' by Francois Chollet

A short summary ...

How to prevent overfitting

- The training of a deep neural net is a long march to fight overfitting

- So far we have these techniques to use:
  1. Choose the right activation function
  2. Use the suitable initialization
  3. Regularize the weights
  4. Add dropout between layers

- What else?

Batch normalization

Normalization...?

- Assume we have this five numbers

$$\mathbf{x} = [1.6, -0.5,\ 0.3,\ 0.4,\ 0.9]$$

- First we check the mean and the standard deviation, and get

$$\mu_{\mathbf{x}} = 0.54 \qquad \sigma_{\mathbf{x}} = 0.69$$

- For each value in the above vector, we perform

$$y_i = \frac{x_i - \mu_{\mathbf{x}}}{\sigma_{\mathbf{x}}}$$

- We get

$$\mathbf{y} = [1.53, -1.50,\ -0.35,\ -0.20,\ 0.52]$$

Source: https://unsplash.com/photos/5IHz5WhosQE/

Batch normalization

Normalization...?

Source: https://unsplash.com/photos/5IHz5WhosQE/

- From

$$\mathbf{x} = [1.6, -0.5,\ 0.3,\ 0.4,\ 0.9]$$

- with the properties

$$\mu_{\mathbf{x}} = 0.54 \qquad \sigma_{\mathbf{x}} = 0.69$$

- to

$$\mathbf{y} = [1.53, -1.50,\ -0.35,\ -0.20,\ 0.52]$$

- What's the big deal? Why do we do that?

- What is the mean and standard deviation of the new vector?

Batch normalization

Normalization

- Normalization / standardization is a process that re-scales data so that it has a zero mean and a standard deviation of 1
- Why do we need to do that in deep learning?

Train and validate on this data set

cat non-cat

But encounter this in actual prediction, what would happen?

Source: https://www.youtube.com/watch?v=nUUqwaxLnWs

Batch normalization

The problem

Source: https://www.youtube.com/watch?v=nUUqwaxLnWs
Source: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

- If an algorithm has learned a mapping from X to Y, and if the distribution of X changes, then the mapping may fail, and we need to re-train to align the distribution of X with the distribution of Y

- In the case of example on cat, what is the mapping?

- In what way the distribution of the training set and the real-world set different?

Batch normalization

The solution

- Let v be the output from a neuron; we denote v_i as one of the neuron outputs from an input in a batch

The batch size is m

$$B = [v_1, v_2, \cdots, v_i, \cdots, v_m]$$

The mini-match mean for v

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} v_i$$

The mini-match variance for v

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (v_i - \mu_B)^2$$

Perform normalization on every item

$$\hat{v}_i \leftarrow \frac{v_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

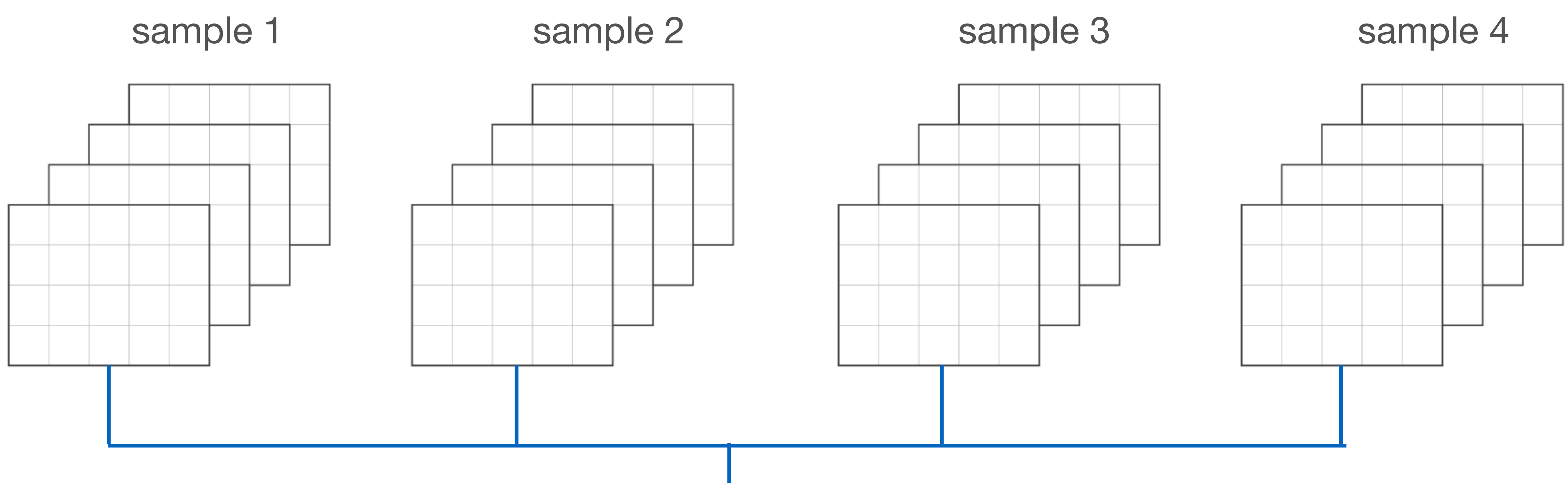
Scale and shift the output, γ and β are trainable parameters; u_i is the normalized output

$$u_i \leftarrow \gamma \hat{v}_i + \beta$$

Batch normalization
on multiple feature maps

- Assume the layer 4 of a particular net generates 4 channel (feature maps), and during traing, the batch size is 4

- The batch normalization is performed channel by channel



Batch normalization is performed channel by channel.
It starts with channel 1. All the elements in channel 1 (of 4 samples) form the v_i and normalized accordingly

Used in Keras

keras.layers.BatchNormalization

- In Keras adding batch normalization is easy
- By default normalization is performed on last axis `(axis = -1)`
- No additional argument is needed (usually)
- Note: the μ and σ are counted as non-trainble parameters, γ and β are trainable parameters

```
> from tensorflow.keras.layers import BatchNormalization

> x    = BatchNormalization()(x)
```

Used in Keras

keras.layers.BatchNormalization

- In many situations, batch normalization is performed before activation

```
> from tensorflow.keras.layers import BatchNormalization
> from tensorflow.keras.layers import Conv2D
> from tensorflow.keras.layers import Activation

> x     = Conv2D(32,(3,3),padding='same')(x)
> x     = BatchNormalization()(x)
> x     = Activation('relu')(x)
```

Time for exercise

Build the model

for cifar 10

- Build the model based on the model plot in the '4_1 Exercise_4_1_model.pdf'

- Note 1: kernel size for all Conv2D is (3,3), padding is 'same', and **no** activation should be set

- Note 2: The dropout value is 0.25 for all Dropout layer, except the last one, which is 0.5

- Note 3: Apply L2 regularization on the last two Conv2D, the value is 0.001

- Note 4: The activation is 'relu' for all Activation layer

- Note 5: The activation for the Dense layer right after the Flatten layer is 'relu'

- Note 6: The activation function for the last layer is 'softmax'

A another short summary ...

How to prevent overfitting

- The training of a deep neural net is a long march to fight overfitting

- So far we have these techniques to use:
  1. Choose the right activation function
  2. Use the suitable initialization
  3. Regularize the weights
  4. Add dropout between layers
  5. Batch normalization

- What else?

A thousand layers

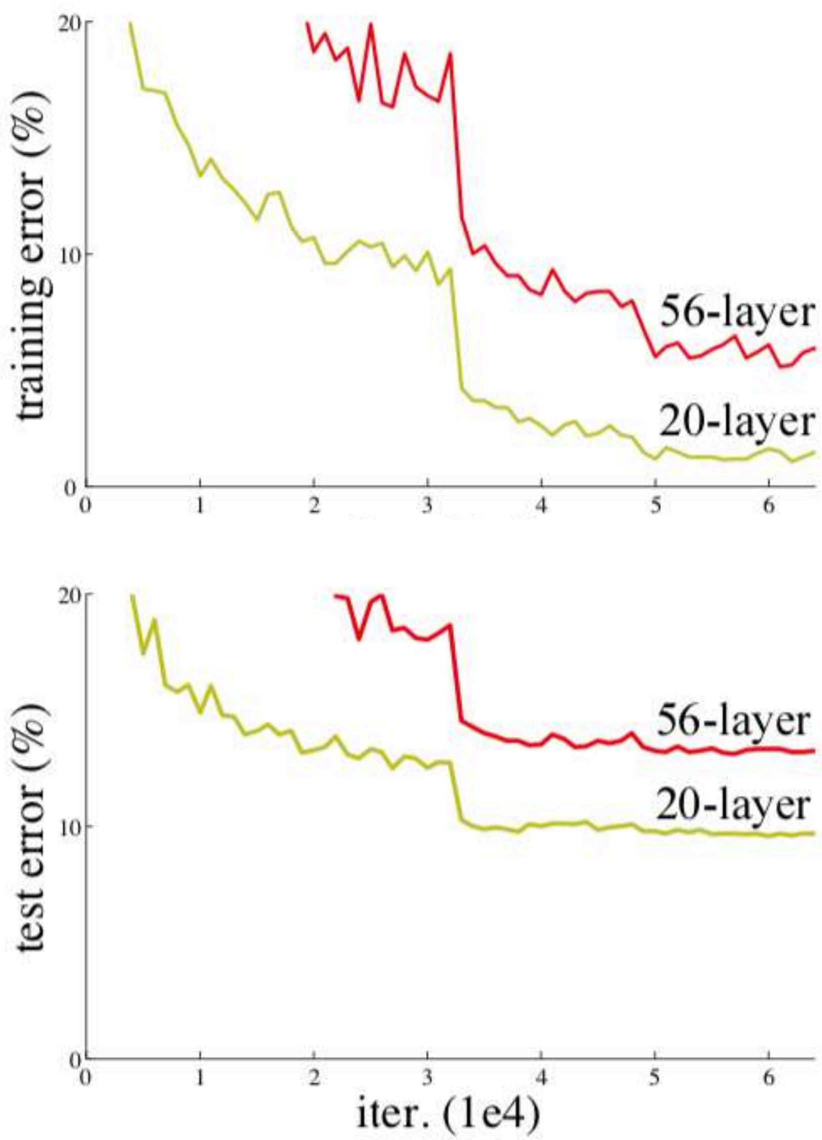
Possible...?

- Techniques available:
  1. Choose the right activation function
  2. Use the suitable initialization
  3. Regularize the weights
  4. Add dropout between layers
  5. Batch normalization

- Given the available techniques, do you think we can train a 1000-layer network?

- Is vanishing gradient still an issue?

Unexpected issue
only in deep deep network

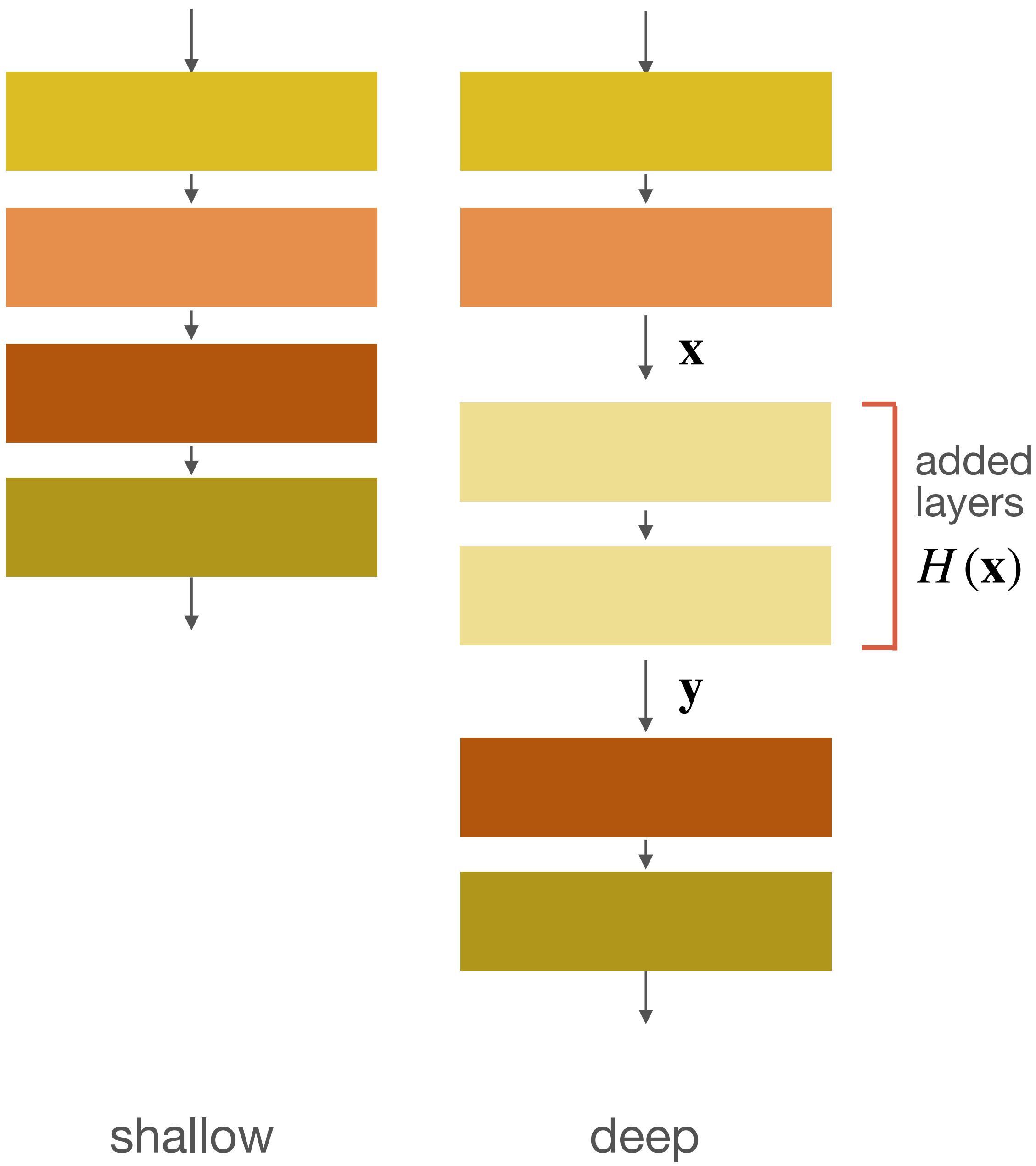


Source: https://arxiv.org/pdf/1512.03385.pdf

- Normalization and proper initialization have somewhat solved the problem of vanishing gradient
- But in very deep network, **degradation** happens: accuracy gets saturated and then degrades
- And, **degradation** *is not caused by overfitting*; the deep model *simply just* has higher training error
- Note: In the case of overfitting, the training error is **lower** (overfit on the training data set) yet the testing error is **higher** (cannot be generalized)

The equal

between shallow and deep



- But deep net should not perform worse than shallow net

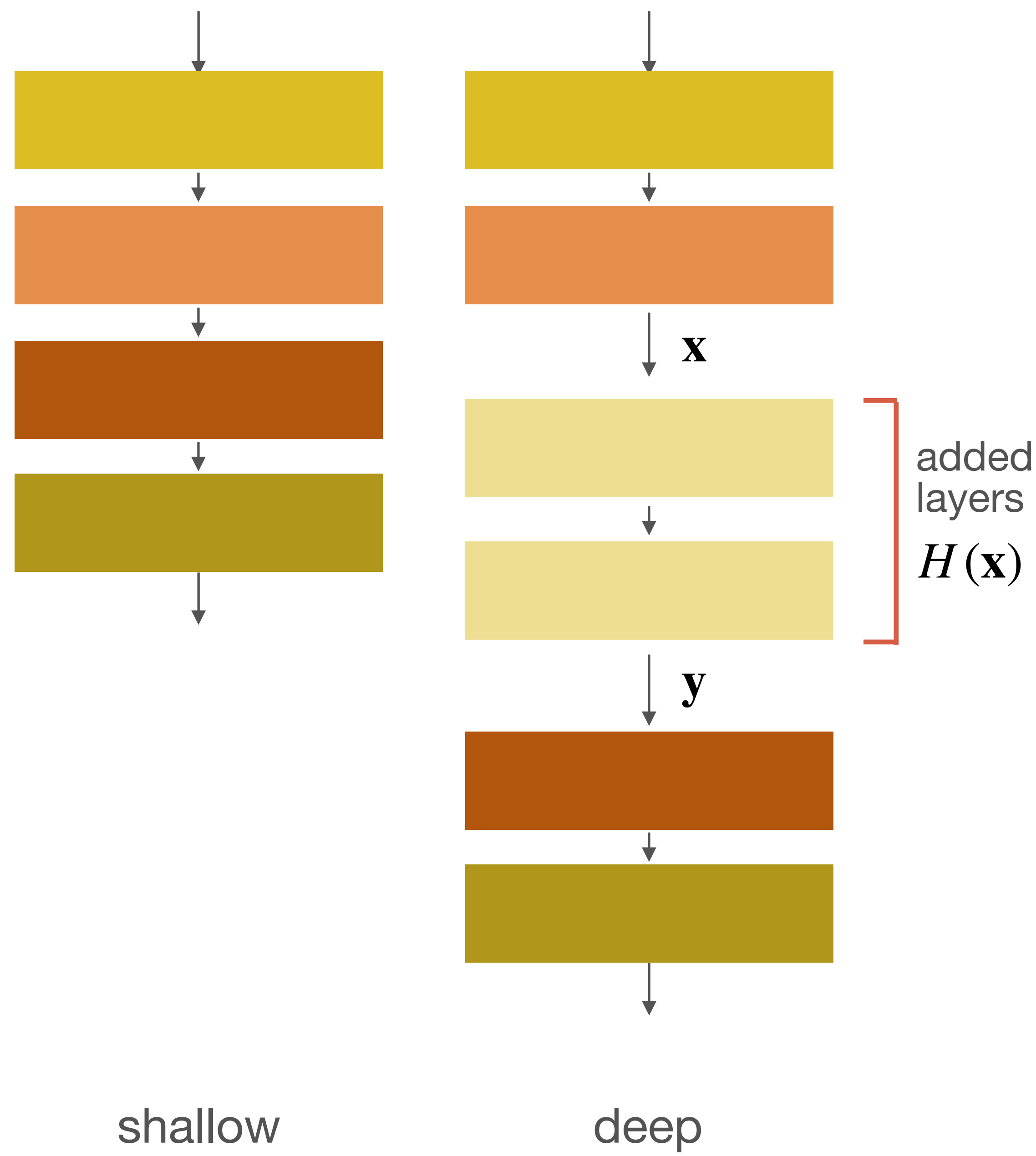
- Why? Assume the added layers simply perform an **identity mapping**, then structure wise, both shallow and deep nets are **in fact the same**

- This implies the layers should perform the below

$$\begin{aligned}\mathbf{y} &= H(\mathbf{x}) \\ &= \mathbf{x}\end{aligned}$$

Conclusion?

between shallow and deep



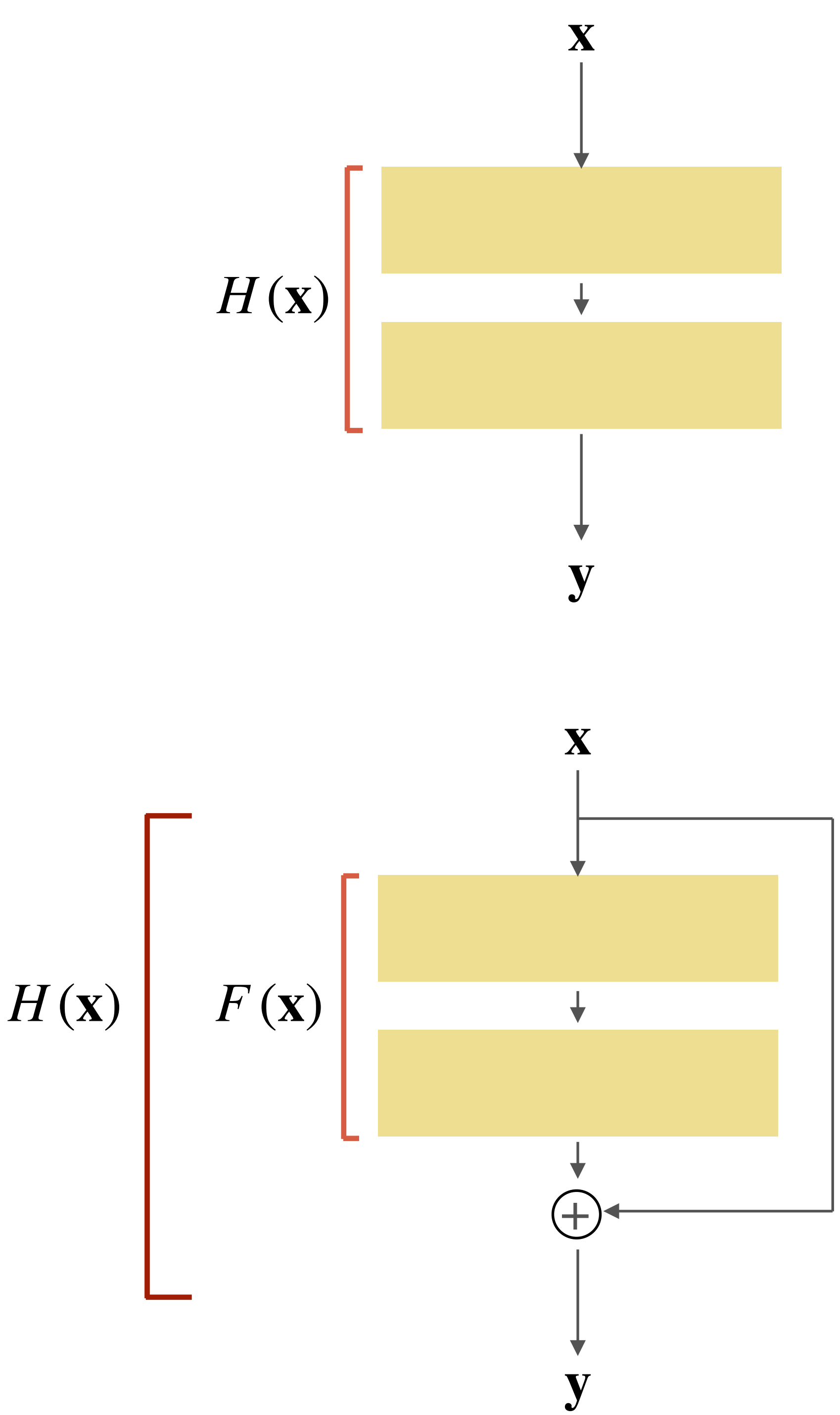
- But since the training error from deeper net is higher, it is safe to say that, the added layers do not perform identity mapping
- Conclusion: it is very hard to get layers trained to perform identity mapping

$$\begin{aligned}\mathbf{y} &= H(\mathbf{x}) \\ &= \mathbf{x}\end{aligned}$$

- But to get the net goes deeper, we need layers that can be trained to at least perform identity mapping, so that it will not perform worse than shallow net
- A solution is needed

The solution

For the depth



- We desire to have an $H(\mathbf{x})$ that can perform identity mapping

$$\begin{aligned} \mathbf{y} &= H(\mathbf{x}) \\ &= \mathbf{x} \end{aligned}$$

- On the left, we know the top structure won't work. He et al. however, proposed that the bottom structure shoud work. Why?

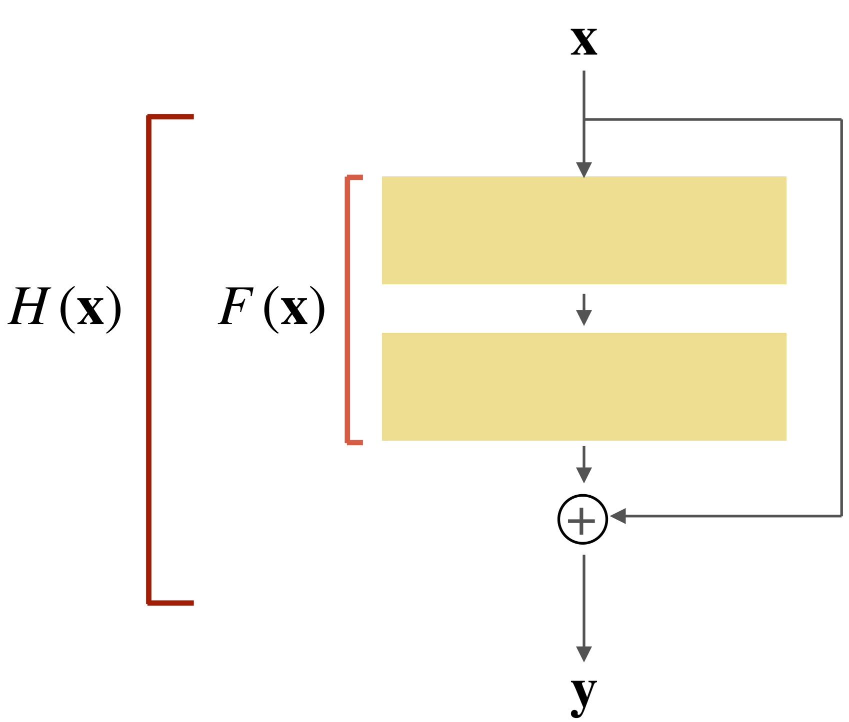
$$\begin{aligned} \mathbf{y} &= H(\mathbf{x}) \\ &= F(\mathbf{x}) + \mathbf{x} \end{aligned}$$

- When $F(\mathbf{x})$ approaches 0, then we have the identity mapping

- It is easier to get non-linear layers pushed to values of 0 rather than values of 1 (remember vanishing gradient?)

Residual layer

The solution



- Since we have

$$\mathbf{y} = H(\mathbf{x}) \\ = F(\mathbf{x}) + \mathbf{x}$$

- Re-arrange the equation, we have

$$H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$$

- Re-arrange the equation again and we get

$$F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$$

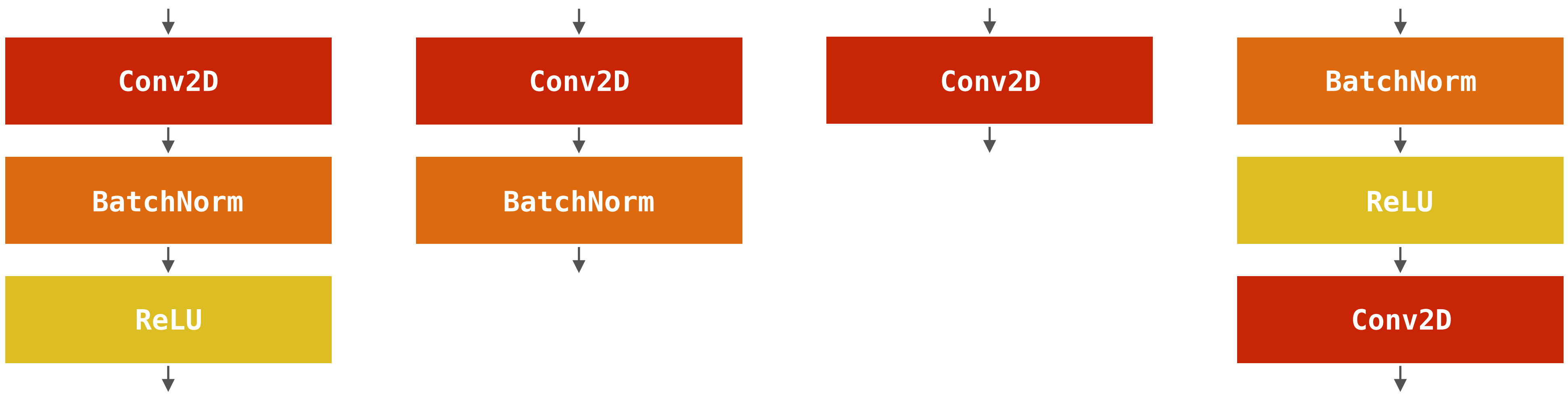
- This implies that $F(\mathbf{x})$ is doing a residual mapping

Let's build the parts and parcel required to form a very deep neural net!

Components in ResNet

Some layers arrangement

- Let's define a few types of layer arrangements
- Note: the Conv2D solely performs convolution with padding, no activation function involves



Define resLyr

Create the function

- Create a function that can encompass the several layer arrangements
- Note: default kernel size is 3 x 3 for 2D convolution

```
> def resLyr(inputs,
             numFilters=16,
             kernelSz=3,
             strides=1,
             activation='relu',
             batchNorm=True,
             convFirst=True,
             lyrName=None):
      convLyr     = Conv2D(numFilters,
                           kernel_size=kernelSz,
                           strides=strides,
                           padding='same',
                           kernel_initializer='he_normal',
                           kernel_regularizer=l2(1e-4),
                           name=lyrName+'_conv' if lyrName else None)
      x           = inputs
      if convFirst:
          x       = convLyr(x)
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,name=lyrName+'_'+activation if lyrName else None)(x)
      else:
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,
                               name=lyrName+'_'+activation if lyrName else None)(x)
          x       = convLyr(x)
      return x
```

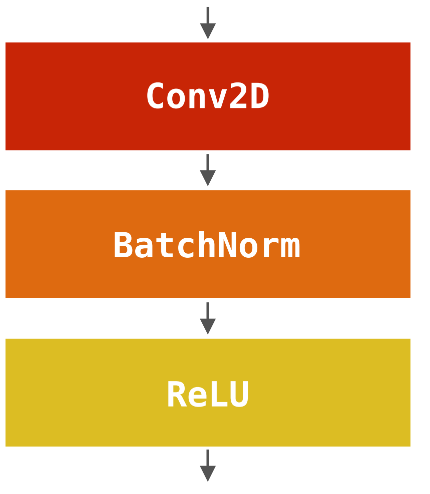
if lyrName is set to 'blk', then the name of the layer will be 'blk_conv', else no name will be given to the layer

resLyr

Types of layer arrangments

```
> def resLyr(inputs,
             numFilters=16,
             kernelSz=3,
             strides=1,
             activation='relu',
             batchNorm=True,
             convFirst=True,
             lyrName=None):
      convLyr     = Conv2D(numFilters,
                           kernel_size=kernelSz,
                           strides=strides,
                           padding='same',
                           kernel_initializer='he_normal',
                           kernel_regularizer=l2(1e-4),
                           name=lyrName+'_conv' if lyrName else None)
      x           = inputs
      if convFirst:
          x       = convLyr(x)
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,name=lyrName+'_'+activation if lyrName else None)(x)
      else:
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,
                               name=lyrName+'_'+activation if lyrName else None)(x)
          x       = convLyr(x)
      return x
```

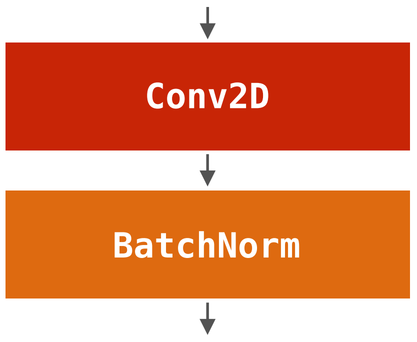
- When
  - `convFirst` is `True`
  - `batchNorm` is `True`
  - `activation` is `'relu'`



resLyr

Types of layer arrangments

- When
  `convFirst` is `True`
  `batchNorm` is `True`
  `activation` is `None`



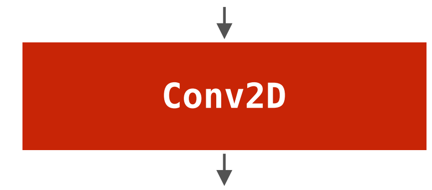

```
> def resLyr(inputs,
             numFilters=16,
             kernelSz=3,
             strides=1,
             activation='relu',
             batchNorm=True,
             convFirst=True,
             lyrName=None):
      convLyr     = Conv2D(numFilters,
                           kernel_size=kernelSz,
                           strides=strides,
                           padding='same',
                           kernel_initializer='he_normal',
                           kernel_regularizer=l2(1e-4),
                           name=lyrName+'_conv' if lyrName else None)
      x           = inputs
      if convFirst:
          x       = convLyr(x)
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,name=lyrName+'_'+activation if lyrName else None)(x)
      else:
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,
                               name=lyrName+'_'+activation if lyrName else None)(x)
          x       = convLyr(x)
      return x
```

resLyr

Types of layer arrangments

```
> def resLyr(inputs,
           numFilters=16,
           kernelSz=3,
           strides=1,
           activation=‘relu’,
           batchNorm=True,
           convFirst=True,
           lyrName=None):
      convLyr     = Conv2D(numFilters,
                           kernel_size=kernelSz,
                           strides=strides,
                           padding='same',
                           kernel_initializer='he_normal',
                           kernel_regularizer=l2(1e-4),
                           name=lyrName+'_conv' if lyrName else None)
      x           = inputs
      if convFirst:
          x       = convLyr(x)
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,name=lyrName+'_'+activation if lyrName else None)(x)
      else:
          if batchNorm:
              x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
          if activation is not None:
              x   = Activation(activation,
                               name=lyrName+'_'+activation if lyrName else None)(x)
          x       = convLyr(x)
      return x
```

- When
  - `convFirst` is `True`
  - `batchNorm` is `False`
  - `activation` is `None`



resLyr

Types of layer arrangments

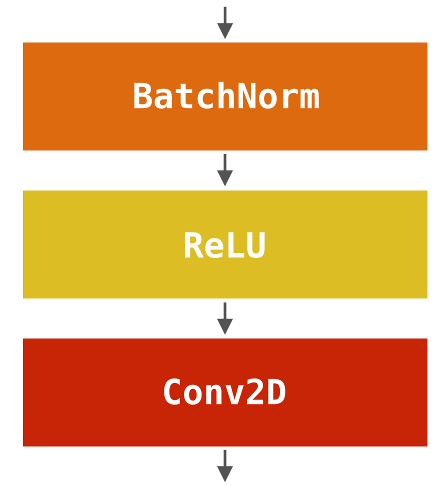
```
> def resLyr(inputs,
           numFilters=16,
           kernelSz=3,
           strides=1,
           activation='relu',
           batchNorm=True,
           convFirst=True,
           lyrName=None):
    convLyr     = Conv2D(numFilters,
                         kernel_size=kernelSz,
                         strides=strides,
                         padding='same',
                         kernel_initializer='he_normal',
                         kernel_regularizer=l2(1e-4),
                         name=lyrName+'_conv' if lyrName else None)
    x           = inputs
    if convFirst:
        x       = convLyr(x)
        if batchNorm:
            x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
        if activation is not None:
            x   = Activation(activation,name=lyrName+'_'+activation if lyrName else None)(x)
    else:
        if batchNorm:
            x   = BatchNormalization(name=lyrName+'_bn' if lyrName else None)(x)
        if activation is not None:
            x   = Activation(activation,
                             name=lyrName+'_'+activation if lyrName else None)(x)
        x       = convLyr(x)
    return x
```

- When

  `convFirst` is `False`

  `batchNorm` is `True`

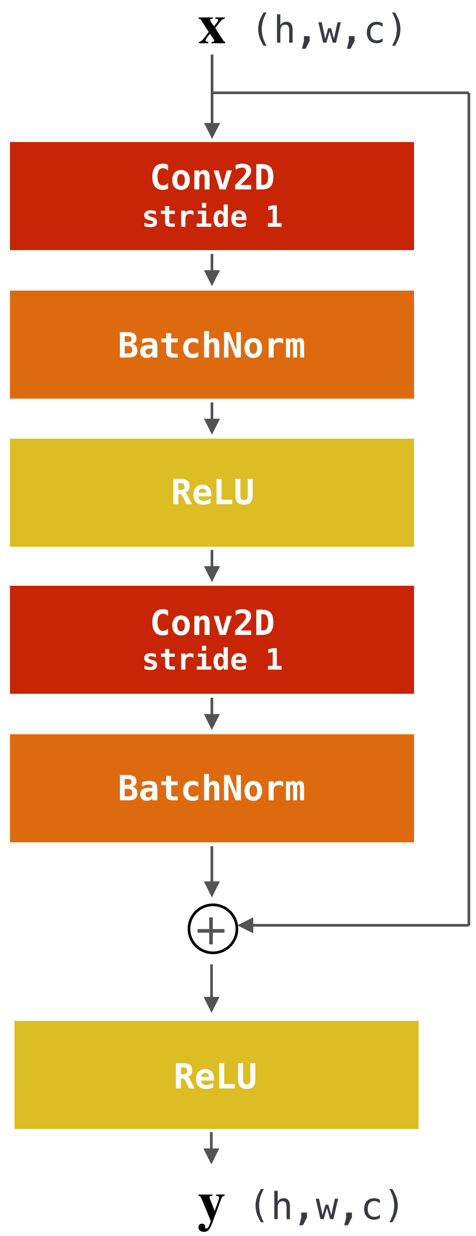
  `activation` is `'relu'`



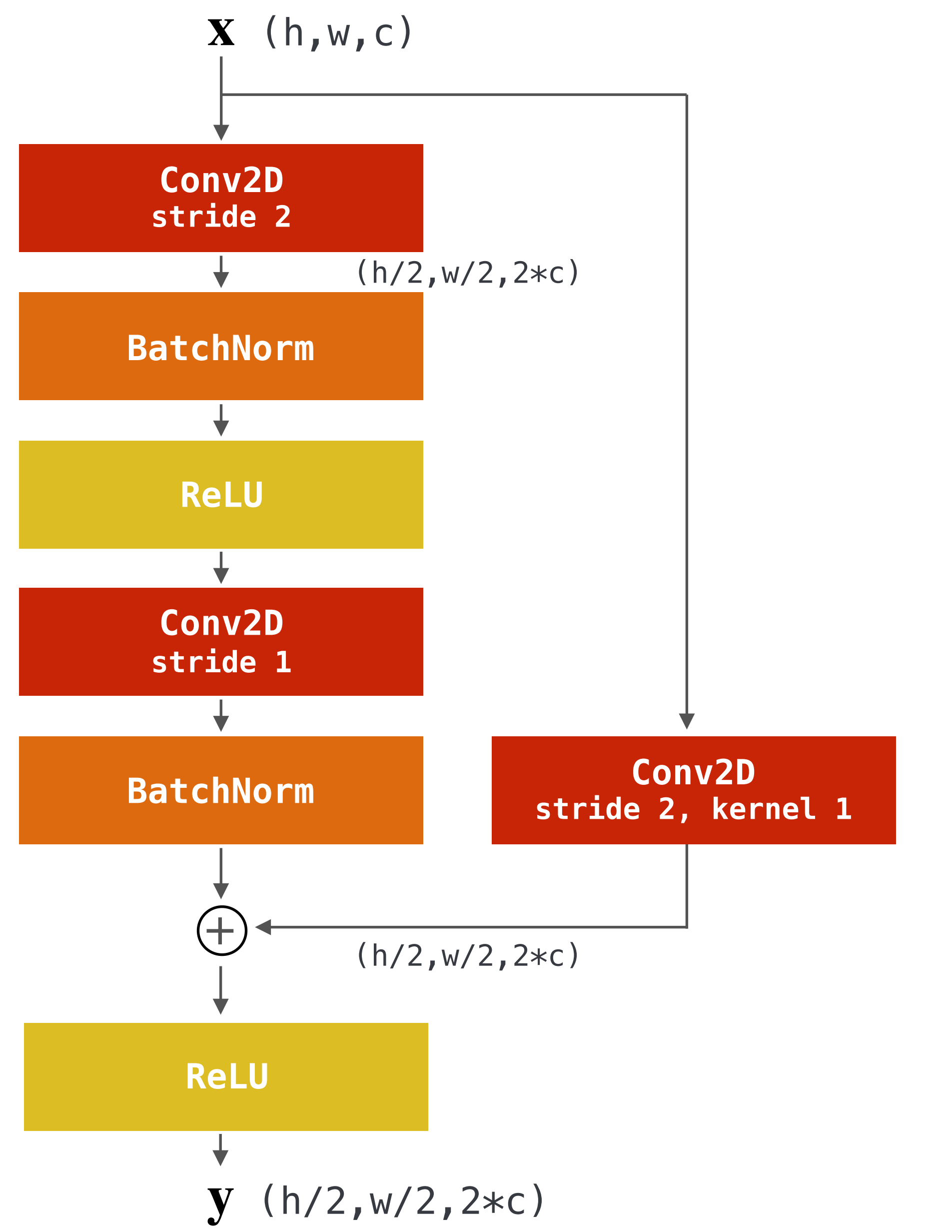
Residual blocks

Mix and match

- Two types of residual blocks in resnet v1



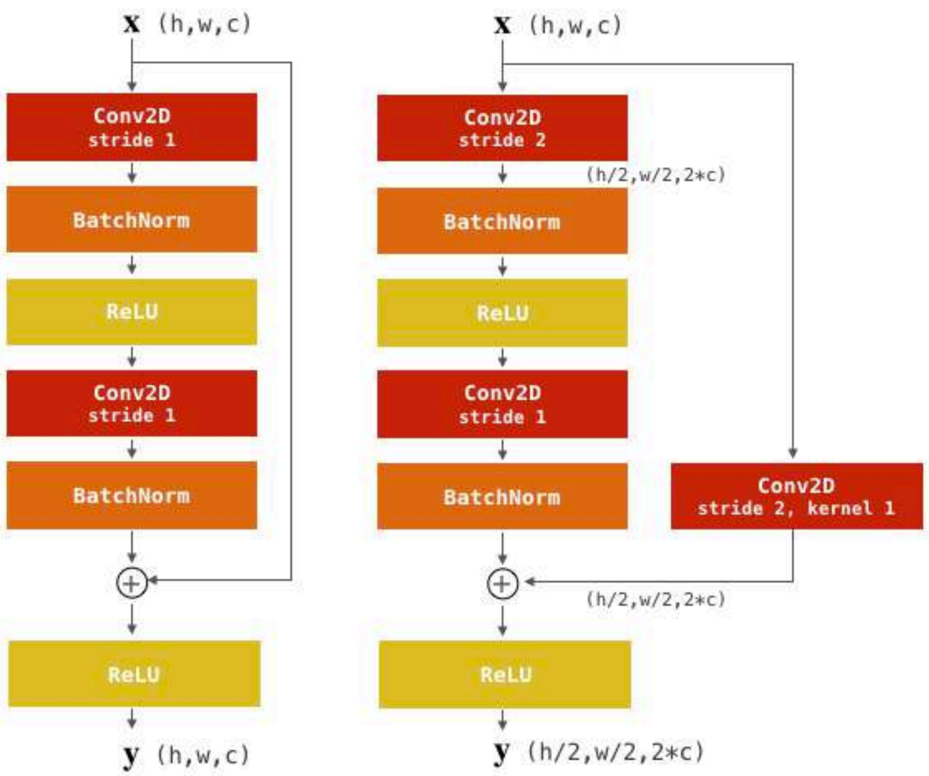
Simp ResBlk



Down ResBlk

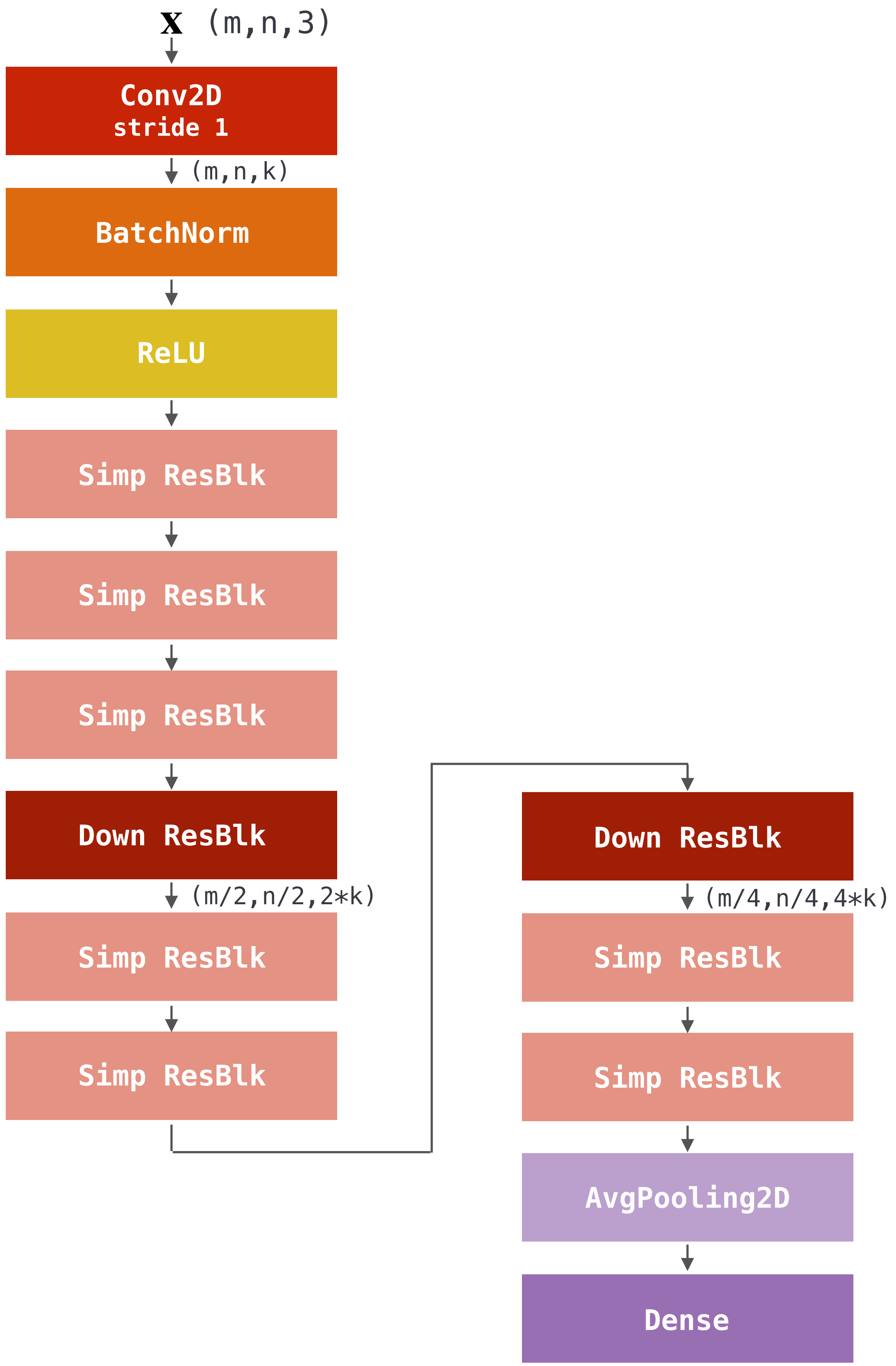
Resnet v1

The full structure



Simp ResBlk

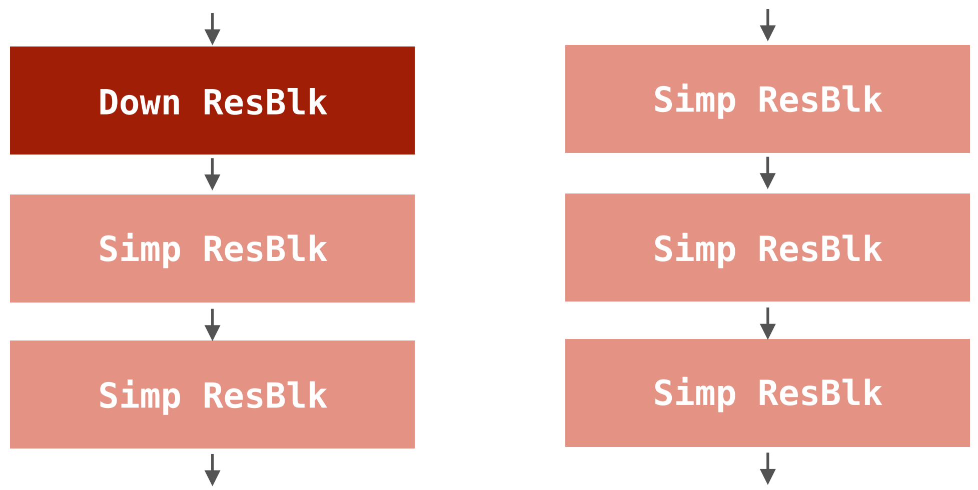
Down ResBlk



Define resBlkV1

```
> def resBlkV1(inputs,
             numFilters=16,
             numBlocks=3,
             downsampleOnFirst=True,
             names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=names+'_Blk'+blkStr+'_Res1' if names else None)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=names+'_Blk'+blkStr+'_Res2' if names else None)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=names+'_Blk'+blkStr+'_lin' if names else None)
        x       = add([x,y],
                      name=names+'_Blk'+blkStr+'_add' if names else None)
        x       = Activation('relu',
                             name=names+'_Blk'+blkStr+'_relu' if names else None)(x)
   return x
```

- The function to produce these two sets of blocks

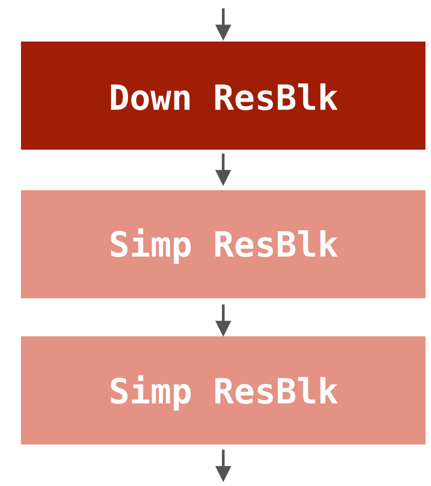


resBlkV1 configuration

```
> def resBlkV1(inputs,
             numFilters=16,
             numBlocks=3,
             downsampleOnFirst=True,
             names=None):
    x        = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=...)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=...)
        if downsampleOnFirst and run == 0:
            x    = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=...)
        x       = add([x,y],
                      name=...)
        x       = Activation('relu',
                             name=...)(x)
    return x
```

- When
  `downsampleOnFirst` is `True`
  `numBlocks` is `3`

- The function creates the below 3 blocks

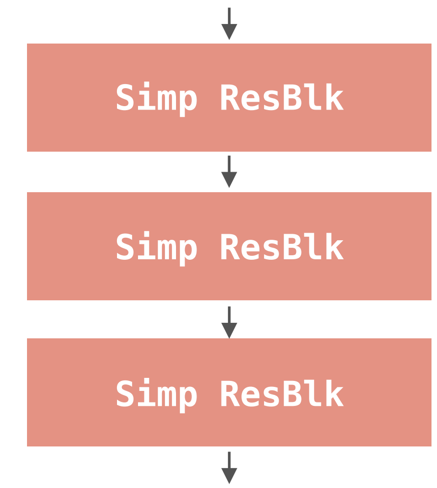


resBlkV1 configuration

```
> def resBlkV1(inputs,
             numFilters=16,
             numBlocks=3,
             downsampleOnFirst=True,
             names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=...)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=...)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=...)
        x       = add([x,y],
                      name=...)
        x       = Activation('relu',
                             name=...)(x)
   return x
```

- When
  `downsampleOnFirst is False`
  `numBlocks         is 3`

- The function create the below 3 blocks

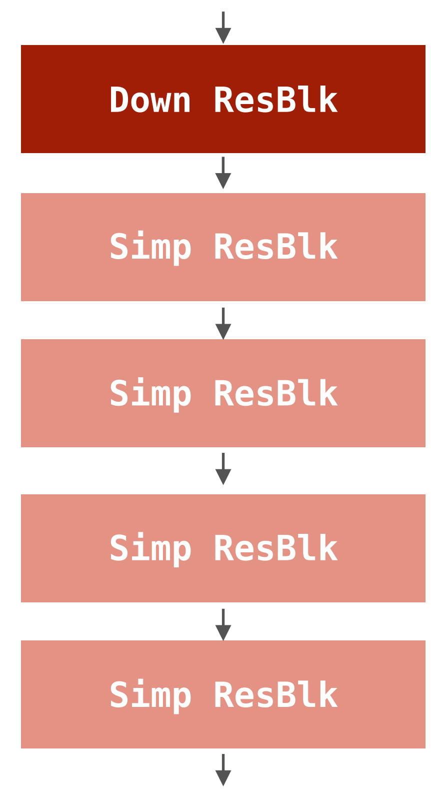


resBlkV1 configuration

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
      x       = inputs
      for run in range(0,numBlocks):
          strides = 1
          blkStr  = str(run+1)
          if downsampleOnFirst and run == 0:
              strides     = 2
          y       = resLyr(inputs=x,
                           numFilters=numFilters,
                           strides=strides,
                           lyrName=...)
          y       = resLyr(inputs=y,
                           numFilters=numFilters,
                           activation=None,
                           lyrName=...)
          if downsampleOnFirst and run == 0:
              x   = resLyr(inputs=x,
                           numFilters=numFilters,
                           kernelSz=1,
                           strides=strides,
                           activation=None,
                           batchNorm=False,
                           lyrName=...)
          x       = add([x,y],
                        name=...)
          x       = Activation('relu',
                               name=...)(x)
     return x
```

- We can have as many blocks as we like; this is what we get when

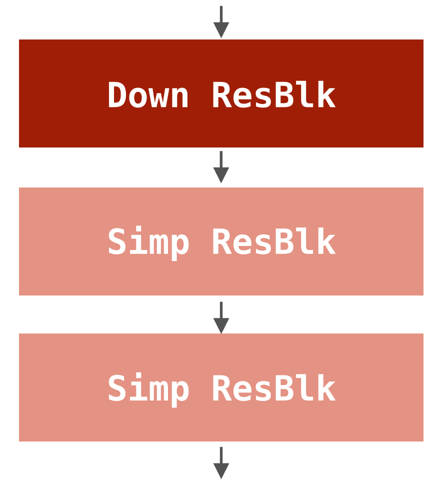
```
downsampleOnFirst is True
numBlocks         is 5
```



resBlkV1 configuration

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
      x        = inputs
      for run in range(0,numBlocks):
          strides = 1
          blkStr  = str(run+1)
          if downsampleOnFirst and run == 0:
              strides     = 2
          y        = resLyr(inputs=x,
                            numFilters=numFilters,
                            strides=strides,
                            lyrName=. . .)
          y        = resLyr(inputs=y,
                            numFilters=numFilters,
                            activation=None,
                            lyrName=. . .)
          if downsampleOnFirst and run == 0:
              x    = resLyr(inputs=x,
                            numFilters=numFilters,
                            kernelSz=1,
                            strides=strides,
                            activation=None,
                            batchNorm=False,
                            lyrName=. . .)
          x        = add([x,y],
                         name=. . .)
          x        = Activation('relu',
                                name=. . .)(x)
     return x
```

- The variable `run` controls the creation of blocks



created when `run == 0` and `downsampleOnFirst == True`

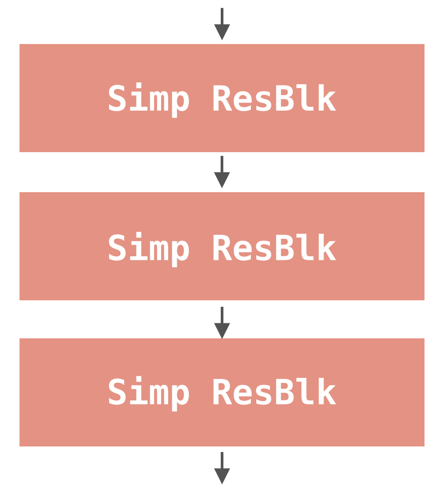
created when `run == 1`

created when `run == 2`

resBlkV1 configuration

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
      x       = inputs
      for run in range(0,numBlocks):
          strides = 1
          blkStr  = str(run+1)
          if downsampleOnFirst and run == 0:
              strides    = 2
          y       = resLyr(inputs=x,
                           numFilters=numFilters,
                           strides=strides,
                           lyrName=. . .)
          y       = resLyr(inputs=y,
                           numFilters=numFilters,
                           activation=None,
                           lyrName=. . .)
          if downsampleOnFirst and run == 0:
              x   = resLyr(inputs=x,
                           numFilters=numFilters,
                           kernelSz=1,
                           strides=strides,
                           activation=None,
                           batchNorm=False,
                           lyrName=. . .)
          x       = add([x,y],
                        name=. . .)
          x       = Activation('relu',
                               name=. . .)(x)
     return x
```

- When `downsampleOnFirst` is set to `False`



created when `run == 0` and `downsampleOnFirst == False`

created when `run == 1`

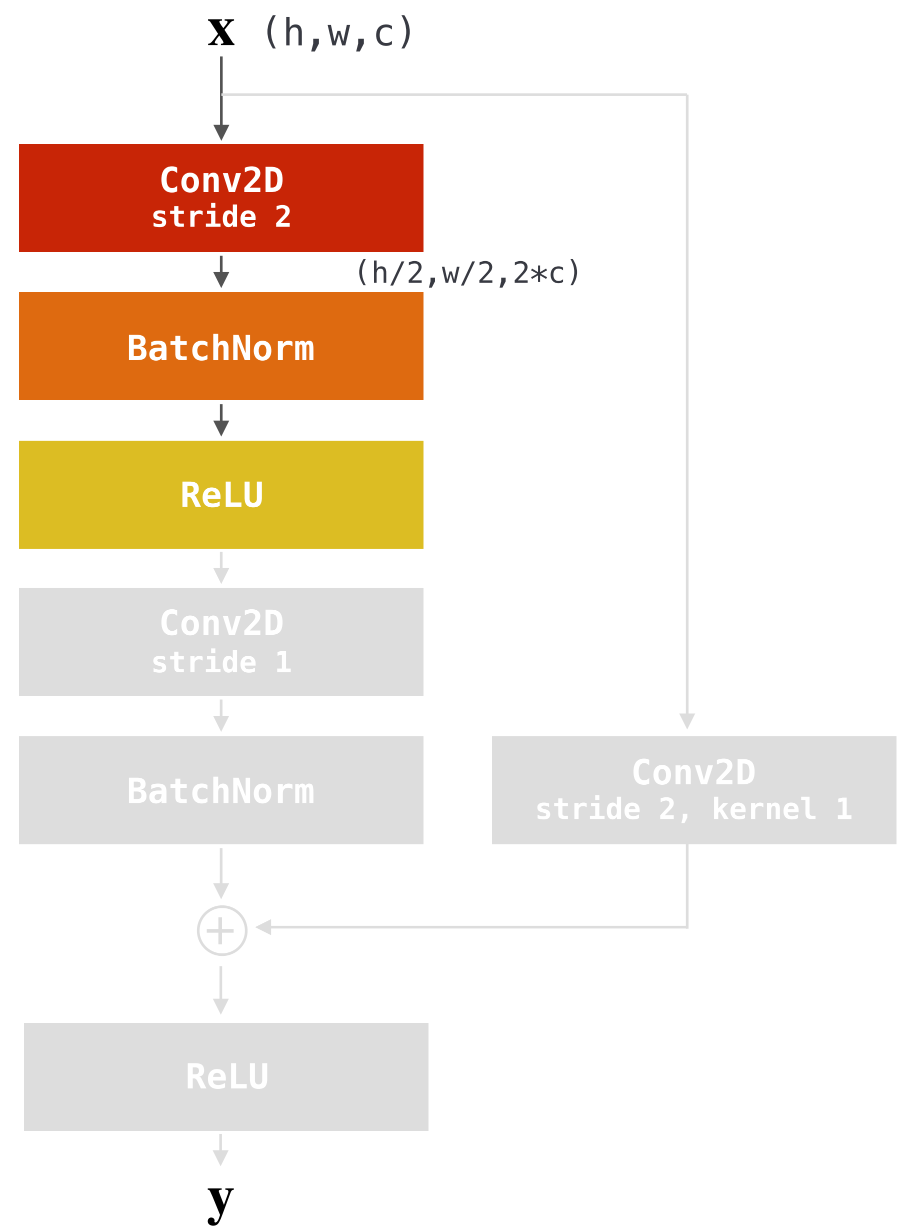
created when `run == 2`

Let's take a look at how 'Down ResBlk' is created

Down ResBlk

```
> def resBlkV1(inputs,
              numFilters=16,
              numBlocks=3,
              downsampleOnFirst=True,
              names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

- Strides is 2 for the first part of the block:

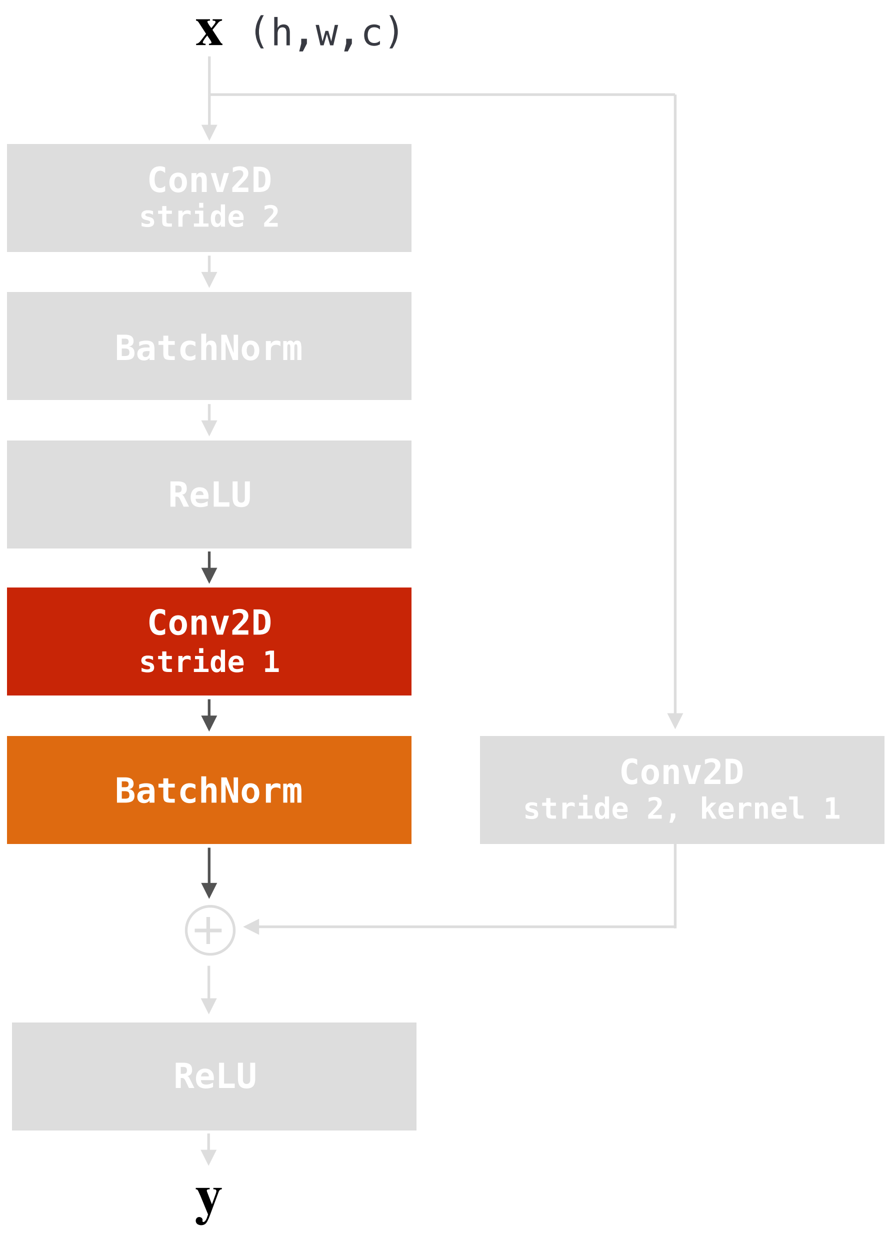


Down ResBlk

Down ResBlk

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
    x        = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x    = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                            name=. . .)(x)
    return x
```

- The second part of the block:

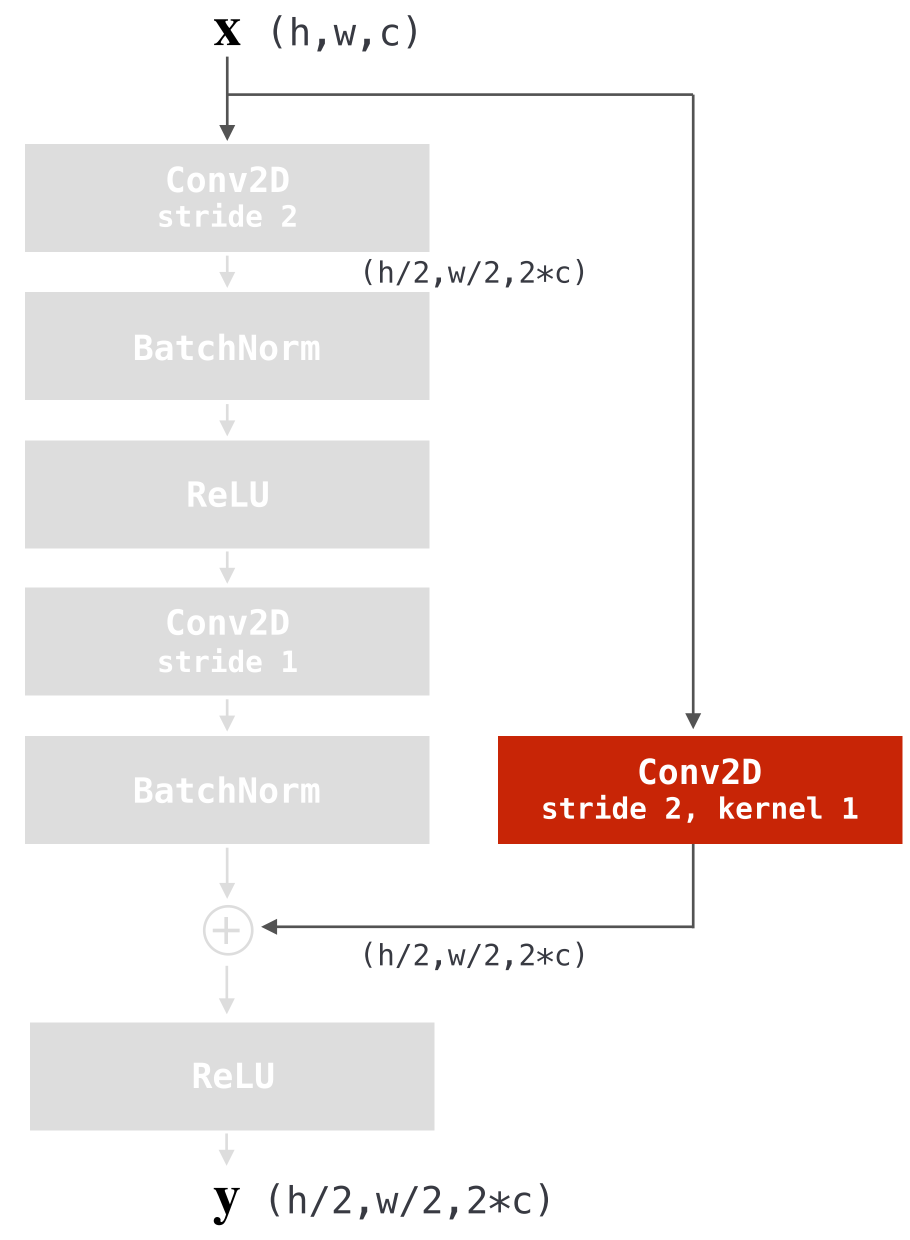


Down ResBlk

Down ResBlk

```
> def resBlkV1(inputs,
              numFilters=16,
              numBlocks=3,
              downsampleOnFirst=True,
              names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

- The third part of the block:

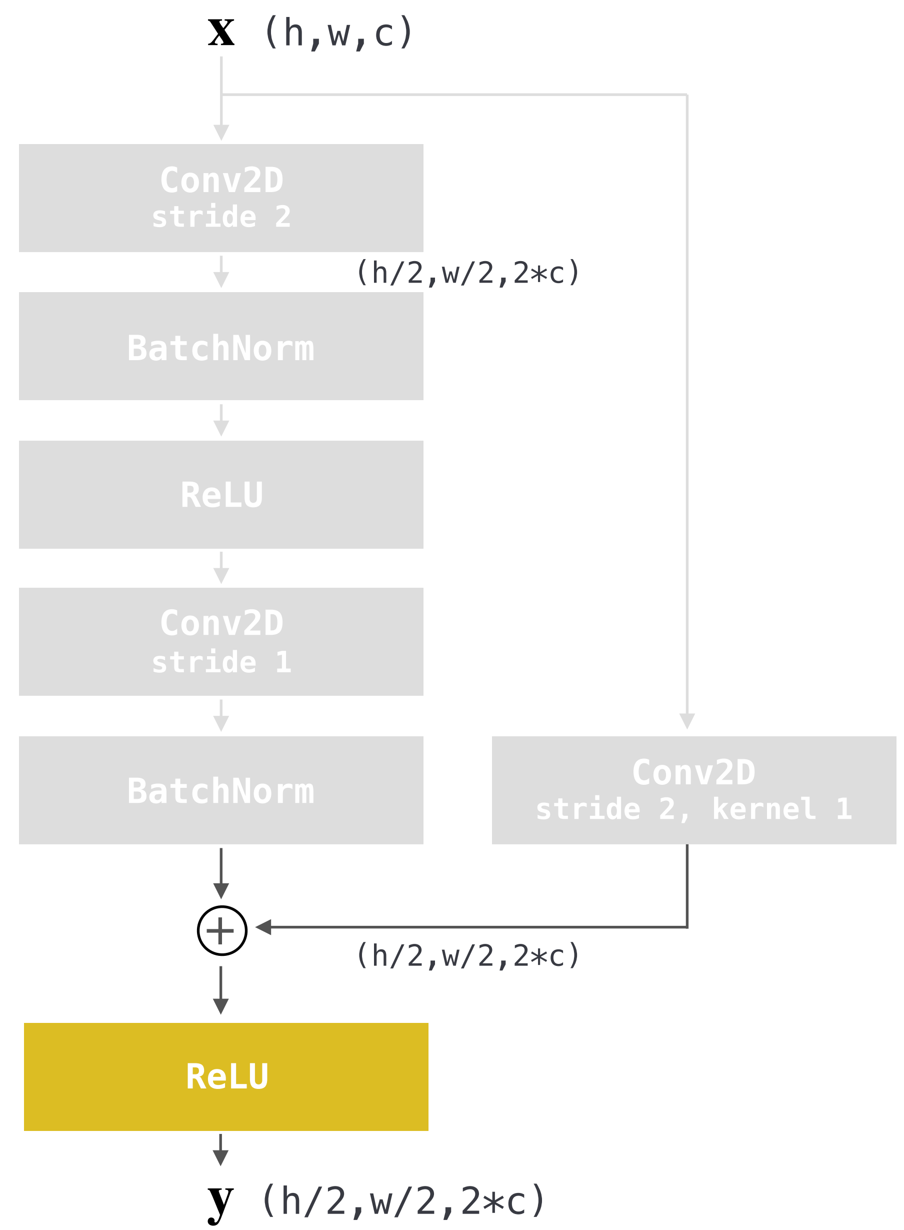


Down ResBlk

Down ResBlk

```
> def resBlkV1(inputs,
             numFilters=16,
             numBlocks=3,
             downsampleOnFirst=True,
             names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

- The last of the block:



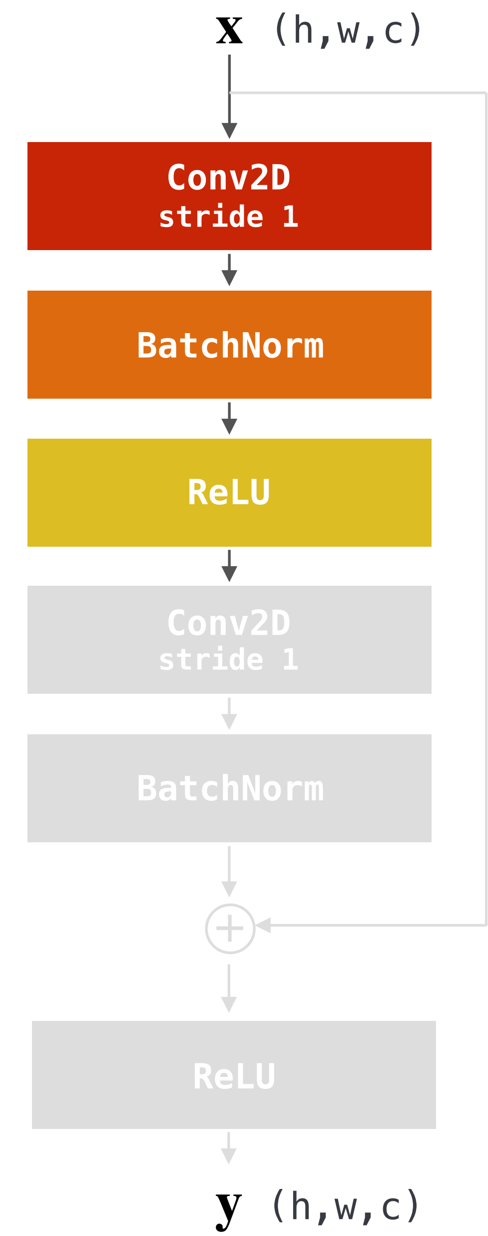
Down ResBlk

How about 'Simp ResBlk'?

Simp ResBlk

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

- Strides is 1 for the first part of the block (no downsampling) :

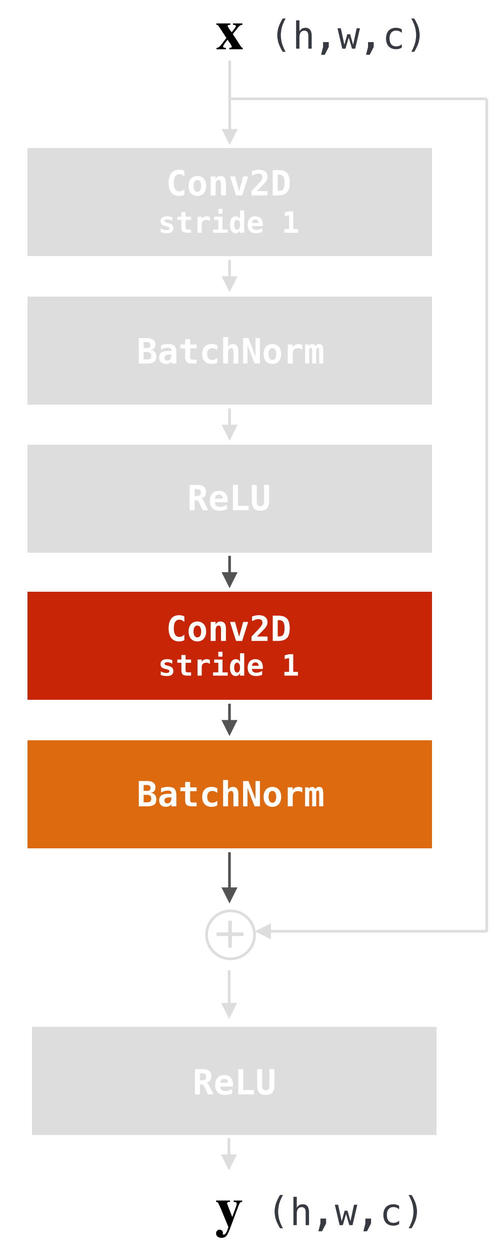


Simp ResBlk

Simp ResBlk

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

- The second part of the block:

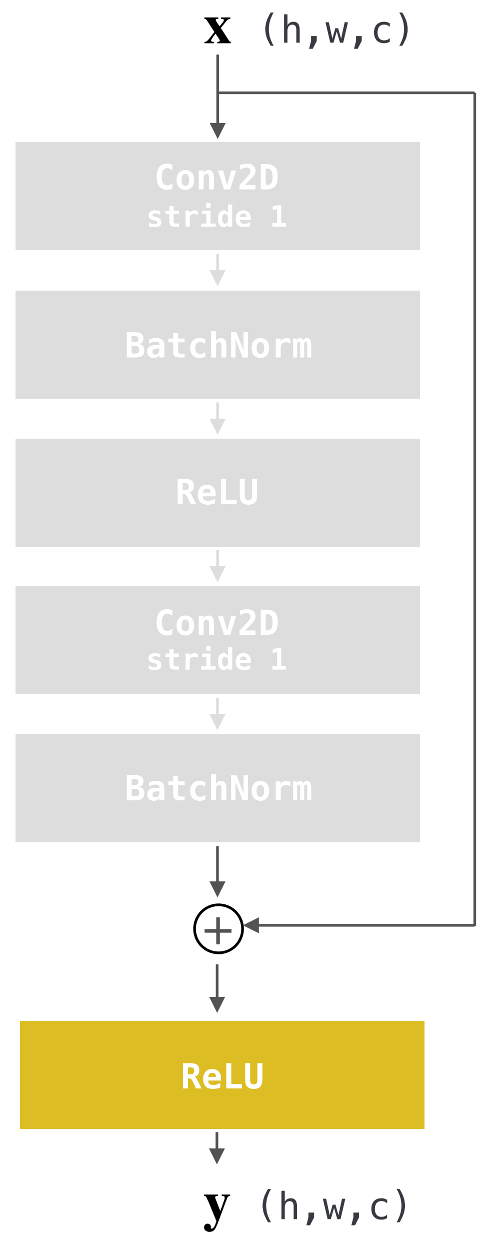


Simp ResBlk

Simp ResBlk

```
> def resBlkV1(inputs,
               numFilters=16,
               numBlocks=3,
               downsampleOnFirst=True,
               names=None):
    x       = inputs
    for run in range(0,numBlocks):
        strides = 1
        blkStr  = str(run+1)
        if downsampleOnFirst and run == 0:
            strides     = 2
        y       = resLyr(inputs=x,
                         numFilters=numFilters,
                         strides=strides,
                         lyrName=. . .)
        y       = resLyr(inputs=y,
                         numFilters=numFilters,
                         activation=None,
                         lyrName=. . .)
        if downsampleOnFirst and run == 0:
            x   = resLyr(inputs=x,
                         numFilters=numFilters,
                         kernelSz=1,
                         strides=strides,
                         activation=None,
                         batchNorm=False,
                         lyrName=. . .)
        x       = add([x,y],
                      name=. . .)
        x       = Activation('relu',
                             name=. . .)(x)
    return x
```

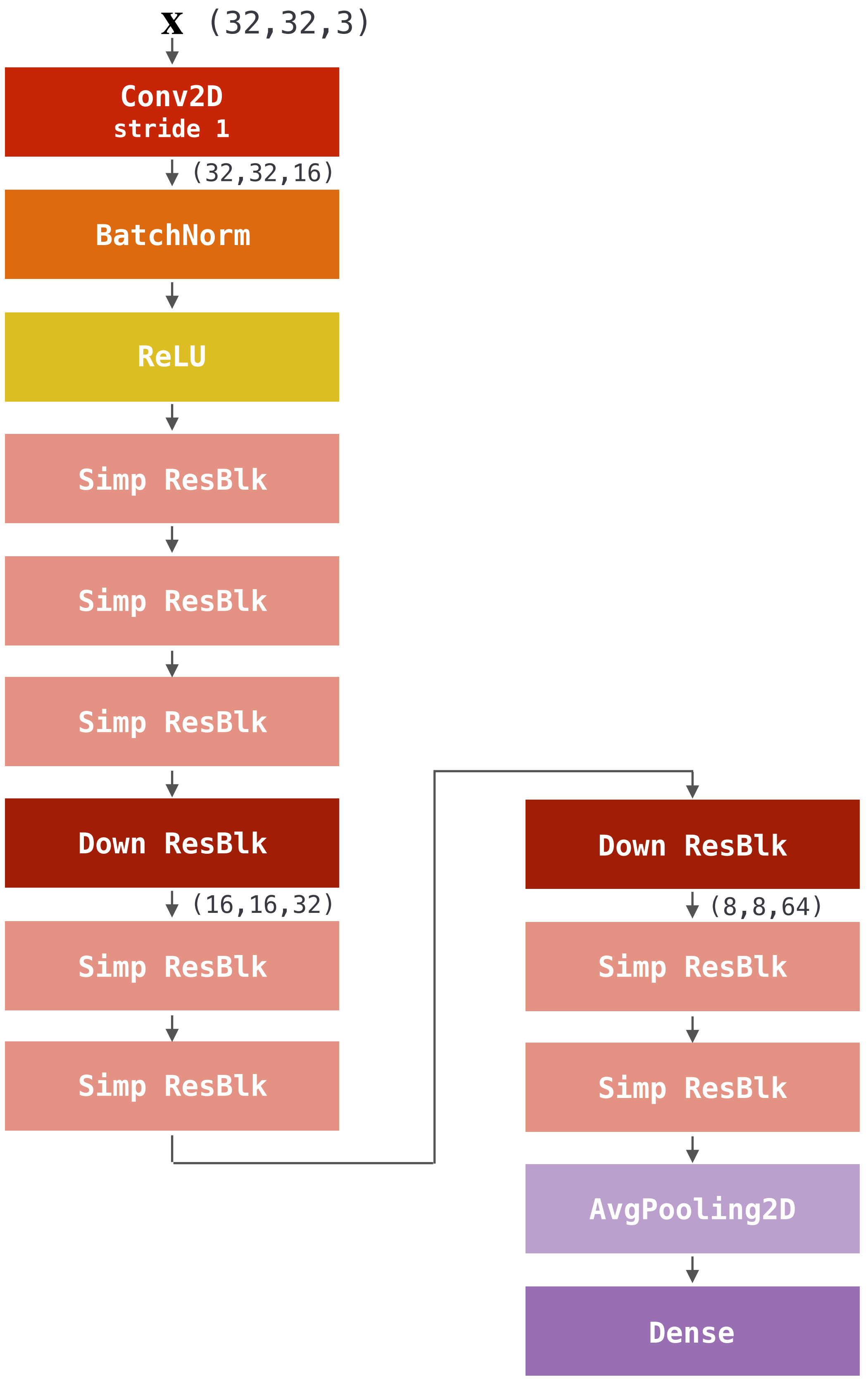
- The last part of the block:



Simp ResBlk

Create full resnet v1

```
> def createResNetV1(inputShape=(32,32,3),
                     numClasses=10):
    inputs       = Input(shape=inputShape)
    v            = resLyr(inputs,
                          lyrName='Inpt')
    v            = resBlkV1(inputs=v,
                            numFilters=16,
                            numBlocks=3,
                            downsampleOnFirst=False,
                            names='Stg1')
    v            = resBlkV1(inputs=v,
                            numFilters=32,
                            numBlocks=3,
                            downsampleOnFirst=True,
                            names='Stg2')
    v            = resBlkV1(inputs=v,
                            numFilters=64,
                            numBlocks=3,
                            downsampleOnFirst=True,
                            names='Stg3')
    v            = AveragePooling2D(pool_size=8,
                                    name='AvgPool')(v)
    v            = Flatten()(v)
    outputs      = Dense(numClasses,
                         activation='softmax',
                         kernel_initializer='he_normal')(v)
    model        = Model(inputs=inputs,outputs=outputs)
    model.compile(loss='categorical_crossentropy',
                  optimizer=optmz,
                  metrics=['accuracy'])

    return model
```



Training
With Adam

- Use Adam as the optimizer.
- The initial learning rate is 0.001
- Set the batch size to be 128

```
> seed      = 29
> np.random.seed(seed)

> optmz     = optimizers.Adam(lr=0.001)
> modelname = 'cifar10ResV1Cfg1'
```

$$m = 0, \quad v = 0$$ Initialization

$$m = \beta_1 \cdot m + \left(1 - \beta_1\right) \cdot dw$$

$$v = \beta_2 \cdot v + \left(1 - \beta_2\right) \cdot dw^2$$

$$\hat{m} = \frac{m}{1 - \beta_1}$$

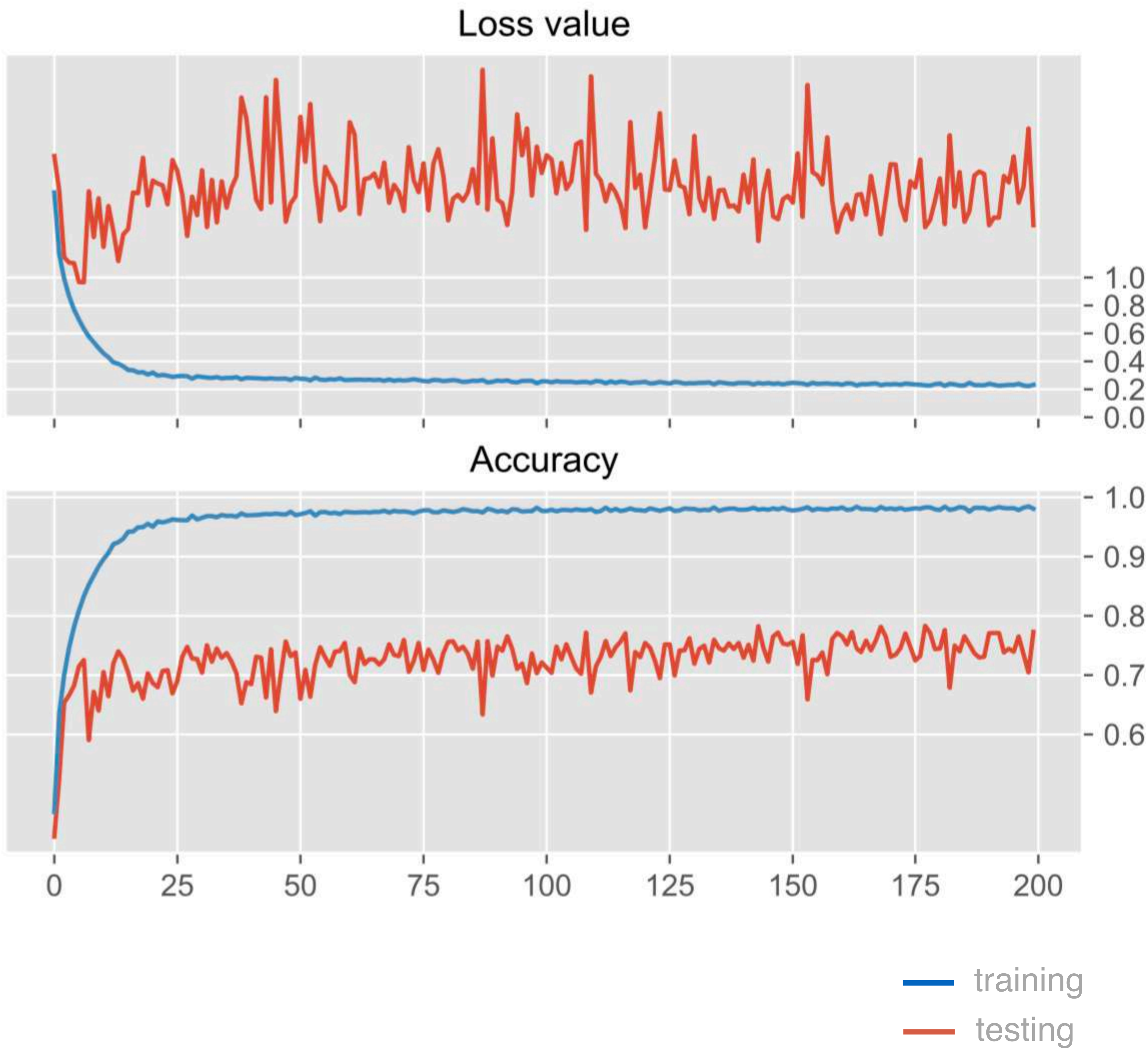
$$\hat{v} = \frac{v}{1 - \beta_2}$$

$$w = w - \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

Source: https://arxiv.org/pdf/1412.6980.pdf

Training

The problem?

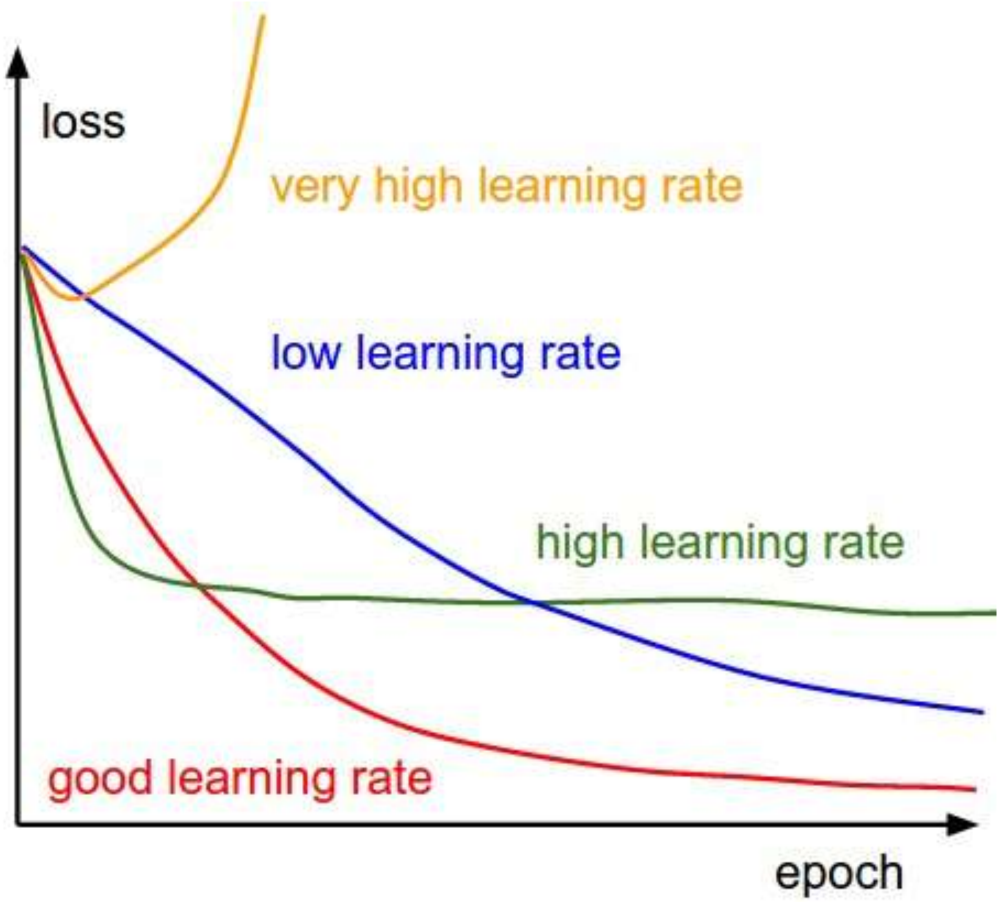


Accuracy: 78.32%

How to unleash the power of a deep net?

Training

Solution to improve?



Source: http://cs231n.github.io/neural-networks-3/

- Learning rate: the amount that the weights are updated during training, also called ‘step size’
- Usually a value between 0.0 and 1.0
- Large learning rate causes model to converge too fast to a suboptimal solution
- Small learning rate gets training stuck

Training

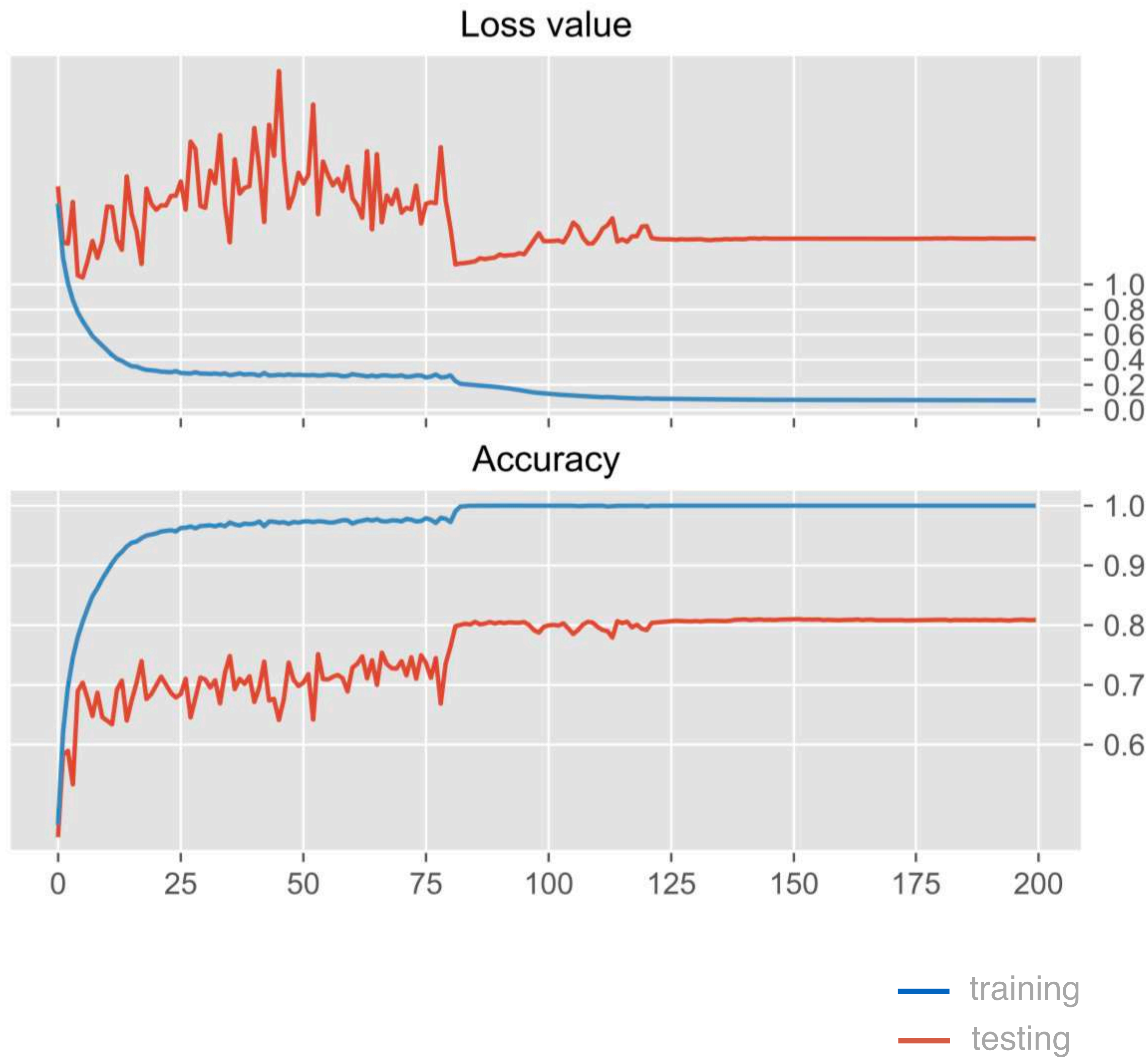
Solution to improve?

- Try learning scheduler

```
> from tensorflow.keras.callbacks import LearningRateScheduler

> def lrSchedule(epoch):
      lr  = 1e-3
      if epoch > 160:
          lr  *= 0.5e-3
      elif epoch > 140:
          lr  *= 1e-3
      elif epoch > 120:
          lr  *= 1e-2
      elif epoch > 80:
          lr  *= 1e-1
      print('Learning rate: ', lr)
      return lr

> LRScheduler     = LearningRateScheduler(lrSchedule)
> callbacks_list  = [checkpoint,csv_logger,LRScheduler]
```

Training

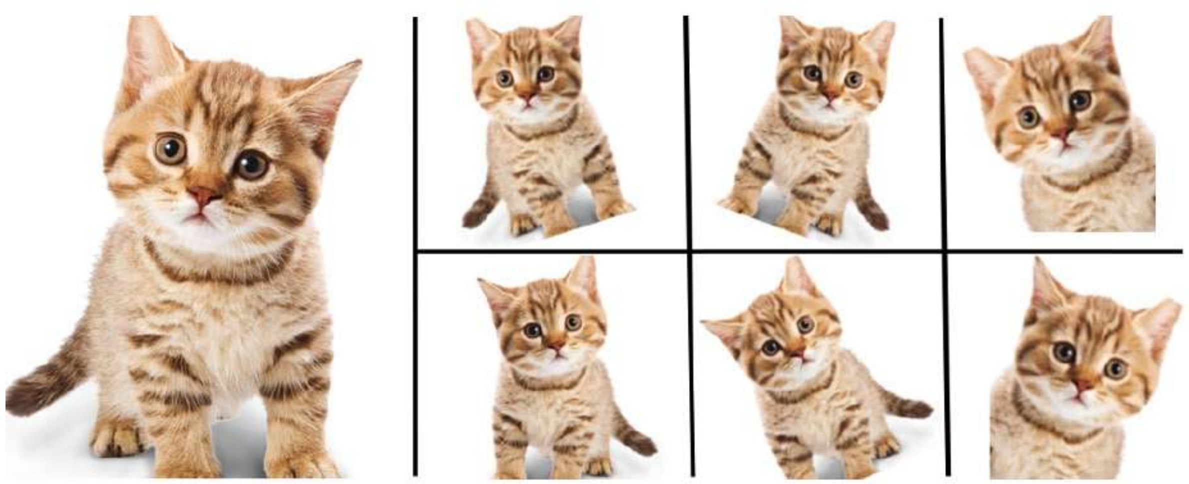
Better?



Accuracy: 81.02%

Training

Solution to improve?



Source: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

- Under current setup, the net sees the same set of images every epoch
- Need to create variety to force the net to learn the features that really matter for classfication
- Generates randomly varied images in the beginning of every epoch, so that the net will not see the exact images twice
- This is called image augmentation

Training
Image augmentation

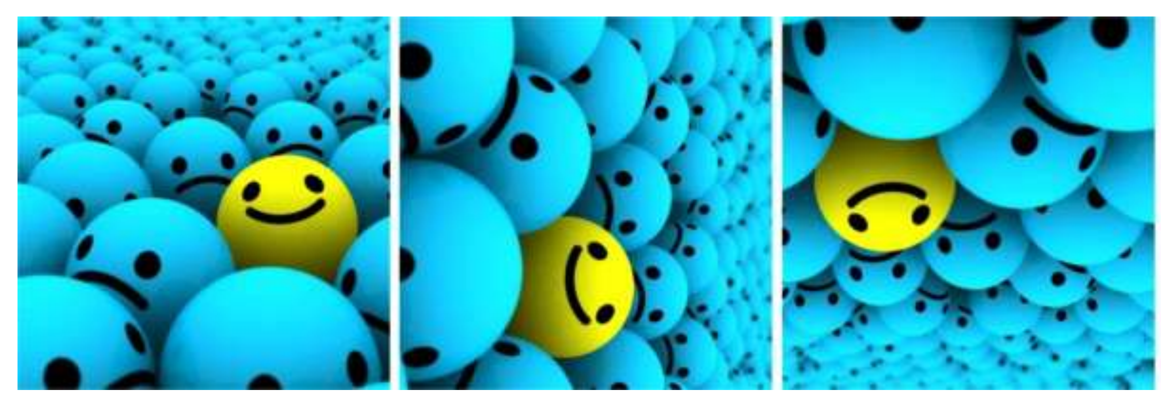
- Types of augmentation

Translation

Zoom

Flip

Rotate



Source: https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

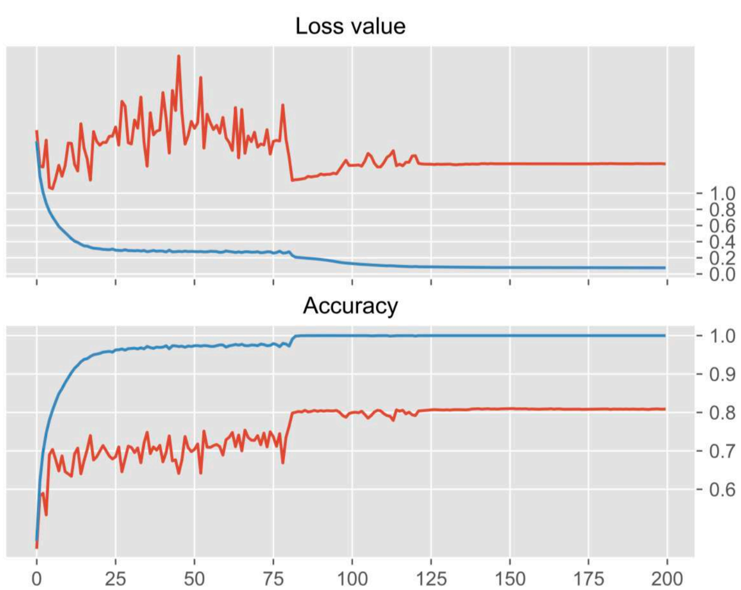
Training
with image generator

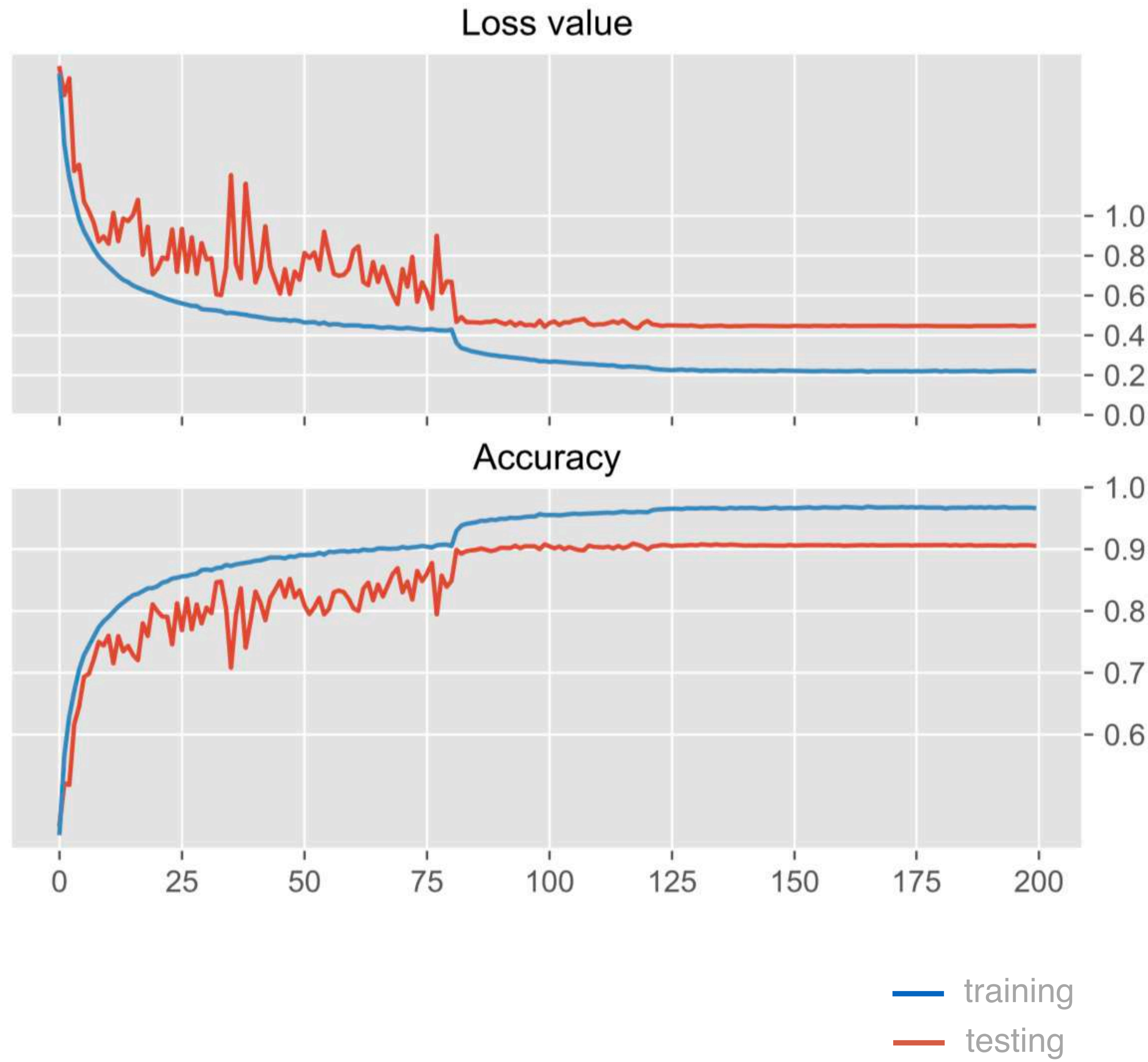
- Build image generator and use fit_generator to train

```
> from tensorflow.keras.preprocessing.image import ImageDataGenerator

> datagen = ImageDataGenerator(width_shift_range=0.1,
                               height_shift_range=0.1,
                               rotation_range=20,
                               horizontal_flip=True,
                               vertical_flip=False)

> model.fit_generator(datagen.flow(trDat, trLbl, batch_size=128),
                      validation_data=(tsDat, tsLbl),
                      epochs=200,
                      verbose=1,
                      steps_per_epoch=len(trDat)/128,
                      callbacks=callbacks_list)
```

Training

This looks good

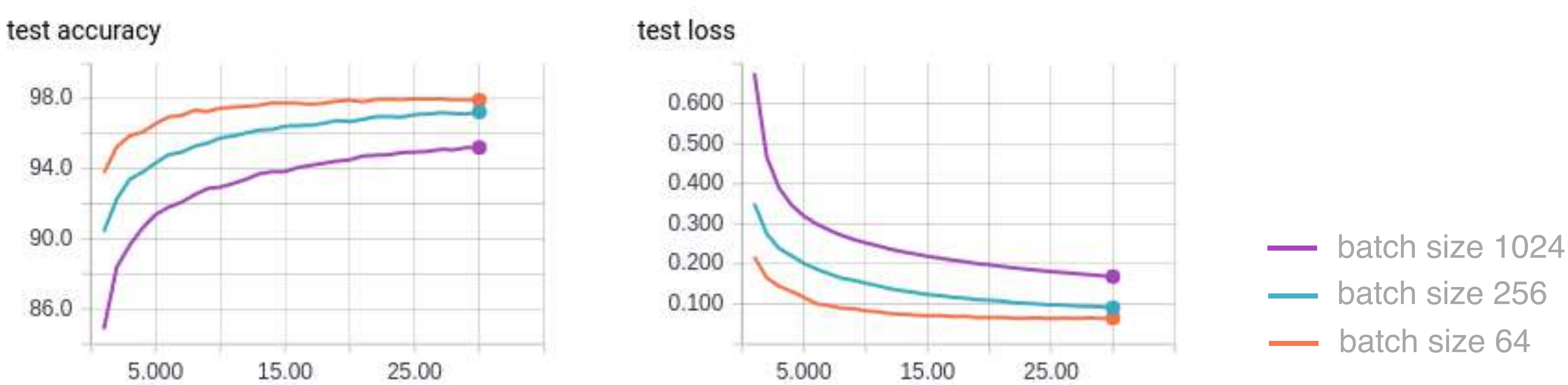




Accuracy: **90.91%**

How about batch size?

- Large batch size: (much) faster training, but accuracy may suffer
- Small batch size: training gets noisy, offering regularizing effect and lower generalization
- GPU may not have sufficient memory to hold large batch of large size



Source: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e

Training
smaller batch size

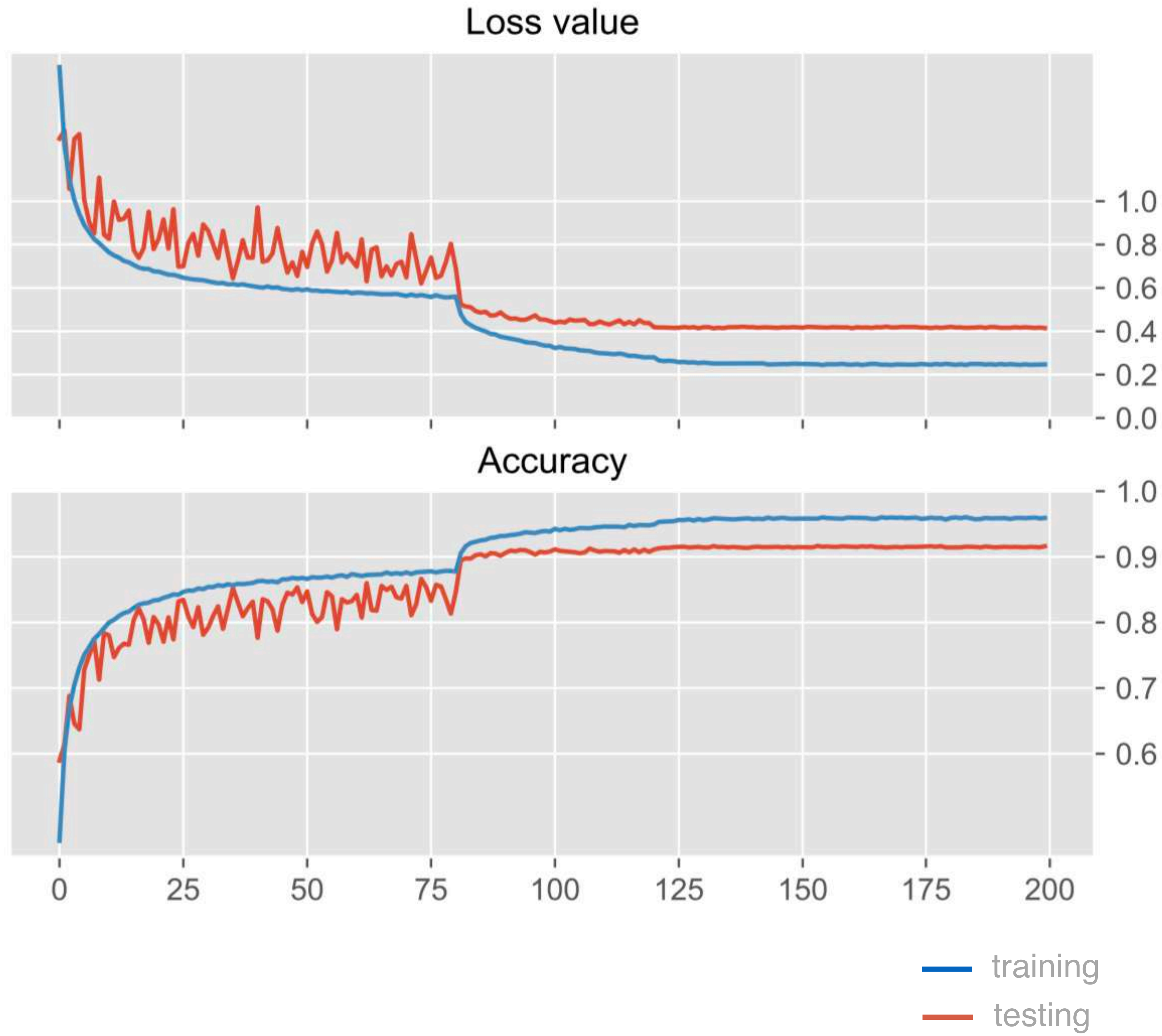
- Make changes on the fit_generator

```
> model.fit_generator(datagen.flow(trDat, trLbl, batch_size=32),
                      validation_data=(tsDat, tsLbl),
                      epochs=200,
                      verbose=1,
                      steps_per_epoch=len(trDat)/32,
                      callbacks=callbacks_list)
```

adjust accordingly

Training

The finale



Accuracy: **91.67%**