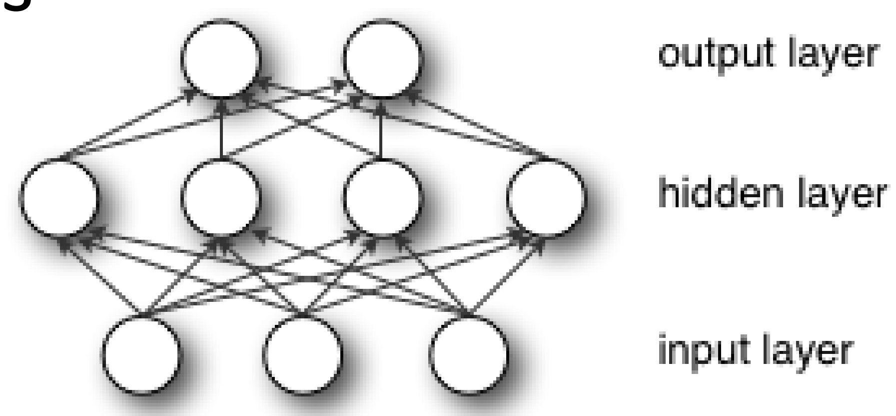
CS109 – Data Science
Deep Learning III - Tips

Hanspeter Pfister, Mark Glickman, Verena Kaynig-Fittkau

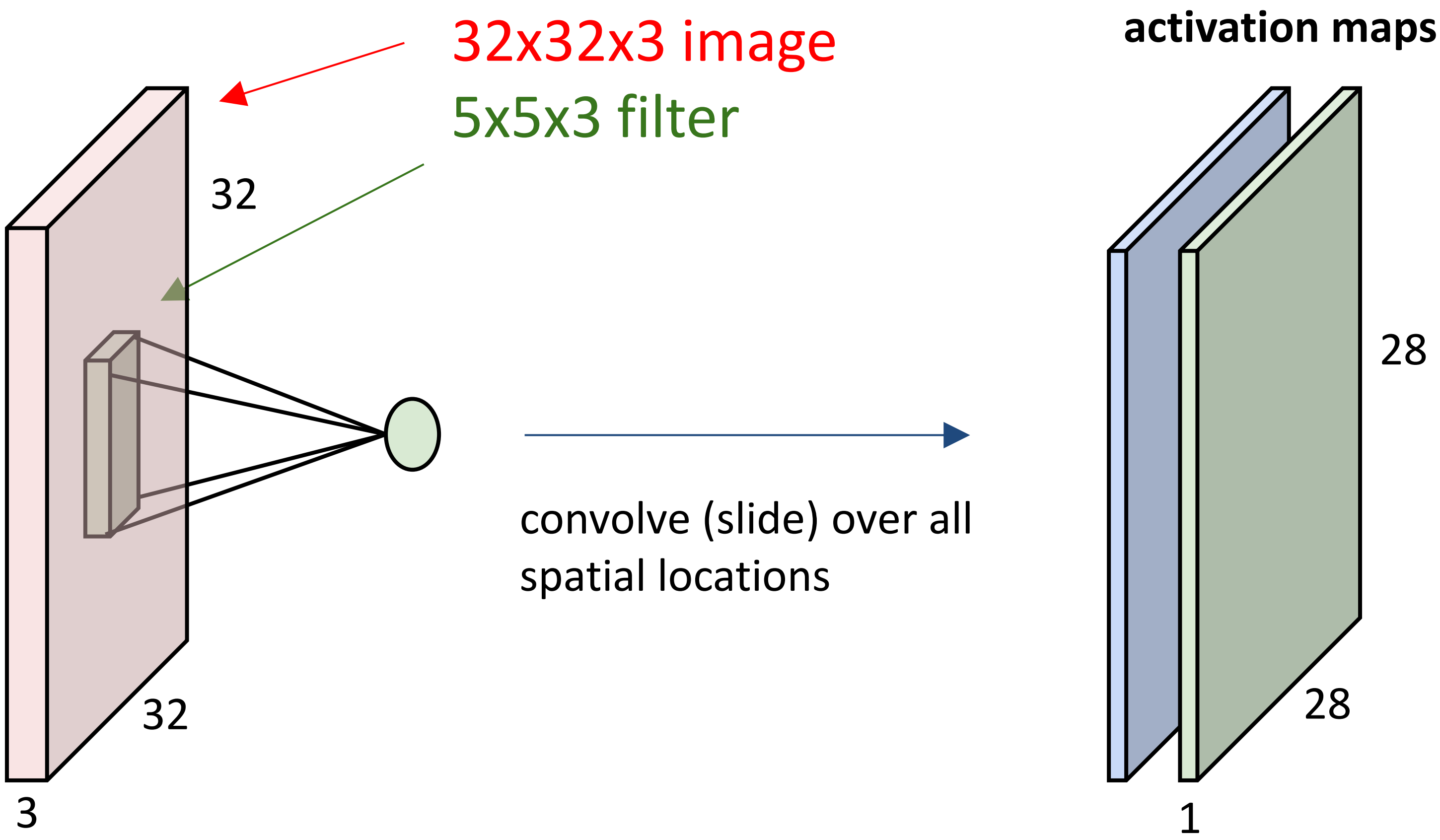
To Train a Simple Network We Need:

- Input layer size
- Number of hidden layers
- Sizes of hidden layers
- Activation function
- Number of output units

- Loss function
- Optimization method

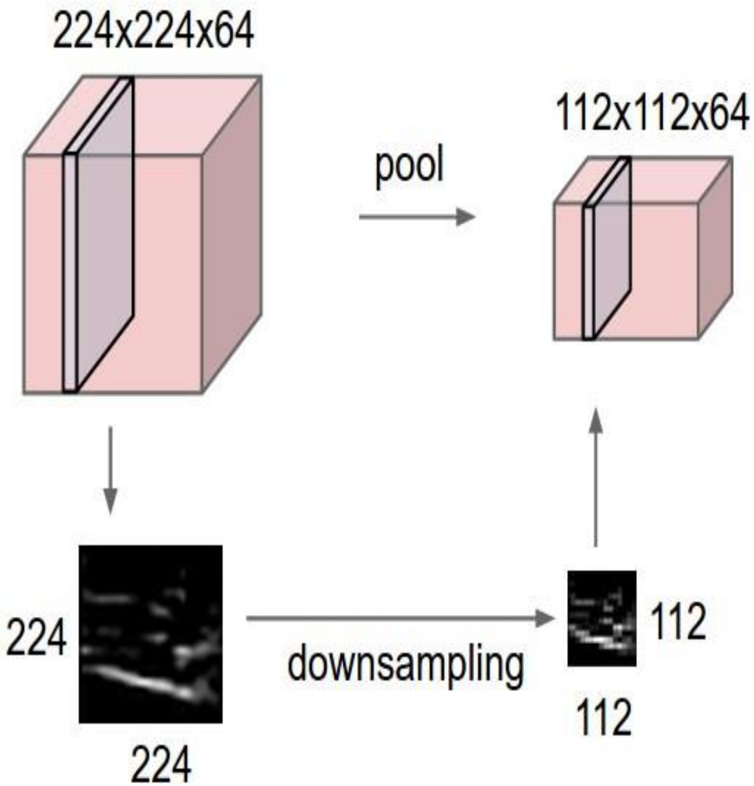


Convolution Layer



http://cs231n.github.io/

Pooling



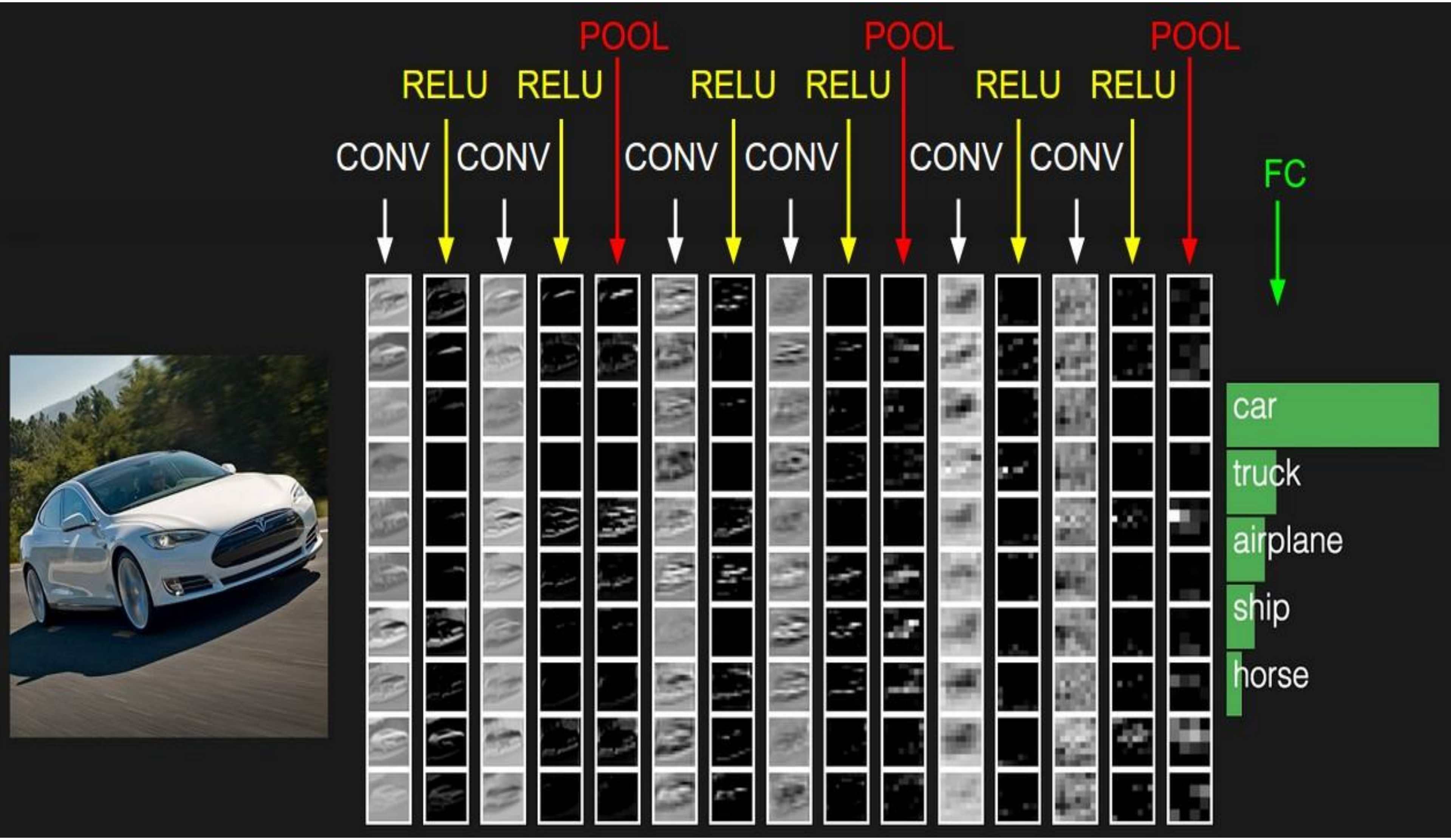
1	1	2	4
5	**6**	7	**8**
3	2	1	0
1	2	3	**4**

2x2 max pooling

6	8
3	4

http://cs231n.github.io/

Typical CNN



http://cs231n.github.io/

Keras Example CNN

Sanity Check

- Try to overfit small portion of training data

- Example:
  - take the first 20 examples from CIFAR-10
  - turn off regularization (reg = 0.0)
  - use simple vanilla ‘sgd’

- If this doesn’t work something is seriously wrong, as in you are using the library incorrectly.

http://cs231n.github.io/

Todays Lecture

Some Essential Things:

- When do you stop training?
- Strategies for learning rate updates
- Dropout and regularization

When to stop training

- Fixed number of epochs
- When the training converged
  - How do you measure convergence?
  - Optimization becomes too slow
  - Validation score doesn't improve
  - Combine with patience counter

Early Stopping in Keras

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss',
                               patience=2)

model.fit(X, y, validation_split=0.2,
          callbacks=[early_stopping])
```

patience: number of epochs with no improvement after which training will be stopped.

More info: https://keras.io/callbacks/

Early Stopping in Keras

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss',
                               patience=2)

model.fit(X, y, validation_split=0.2,
          callbacks=[early_stopping])
```

validation_split: fraction of training data used for validation. Data is split from the end of the data and remains the same during training.

More info: https://keras.io/callbacks/

Learning Rate

- So many options:
- Fixed learning rate
- Start with large LR for n epochs, then set to smaller value
- Slowly decay learning rate over time
- Start with large learning rate, when patience counter expires lower the rate, repeat.

Learning Rate Decay in Keras:

```
def step_decay(epoch):
    lrate = 0.1
    if epoch > 100:
        lrate = 0.1 / (2. ** epoch)
    return lrate

lr = LearningRateScheduler(step_decay)
callbacks_list = [lr]
```

schedule: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

Reduce LR on Plateau

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                              factor=0.2,
                              patience=5,
                              min_lr=0.001)

model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

factor: How much to reduce the learning rate

patience: How many epochs with no improvement until lr is updated

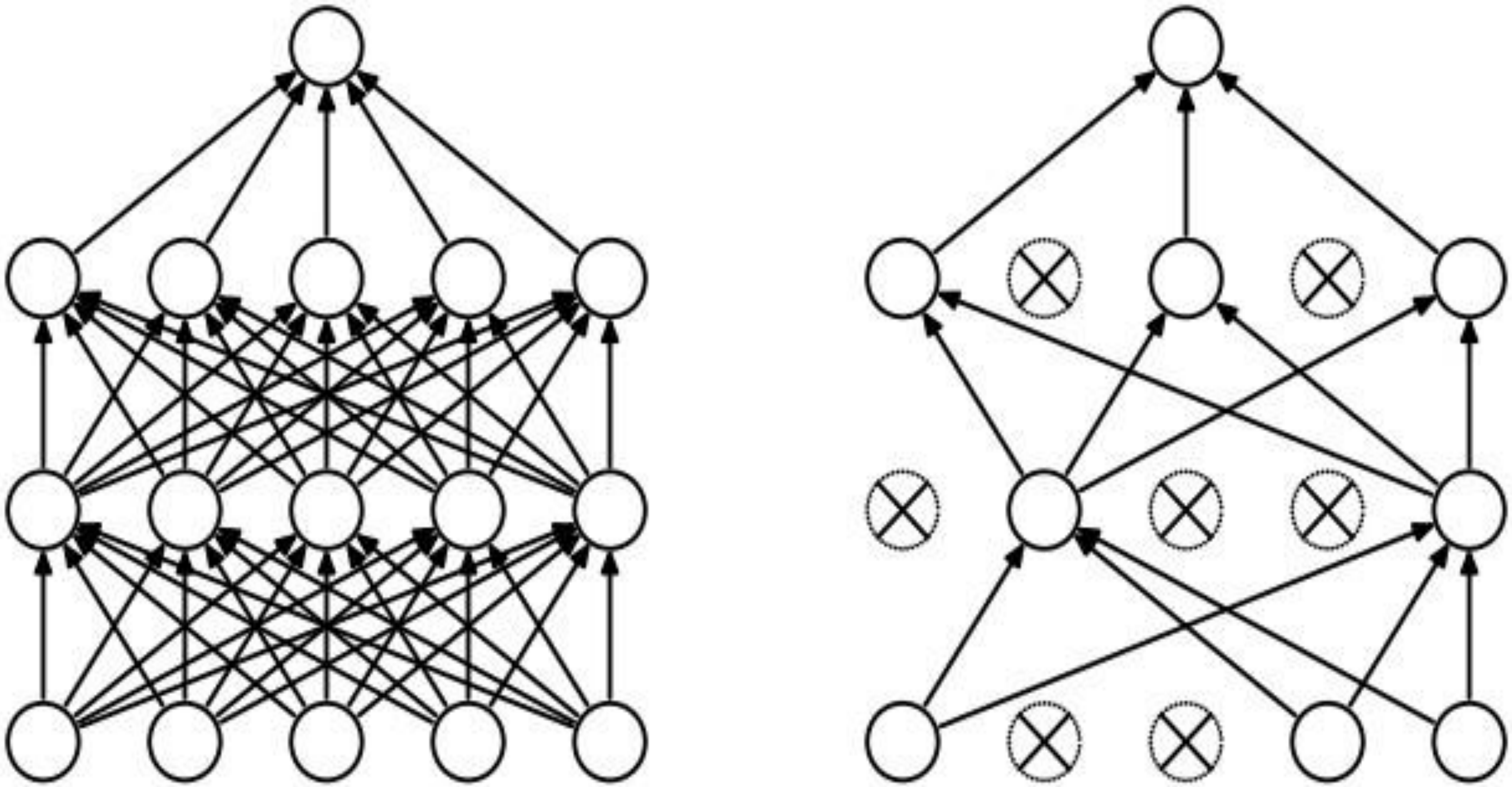
min_lr: When to completely stop

Regularization: dropout

- We have seen that randomness can be used very effectively for regularization
- Remember Random forest
- Can we do something similar for deep learning?

Regularization: **Dropout**

"randomly set some neurons to zero"



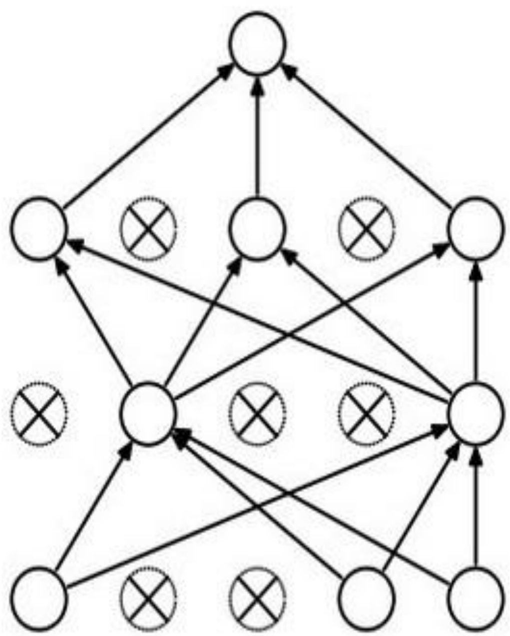
(a) Standard Neural Net   (b) After applying dropout.

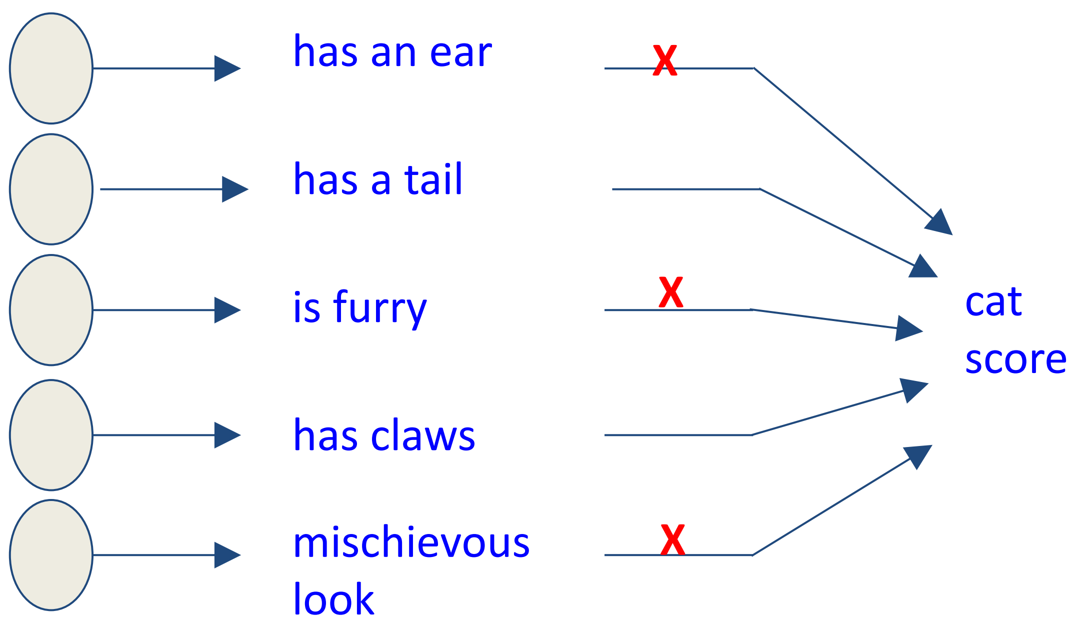
[Srivastava et al., 2014]

http://cs231n.github.io/

How could this possibly be a good idea?

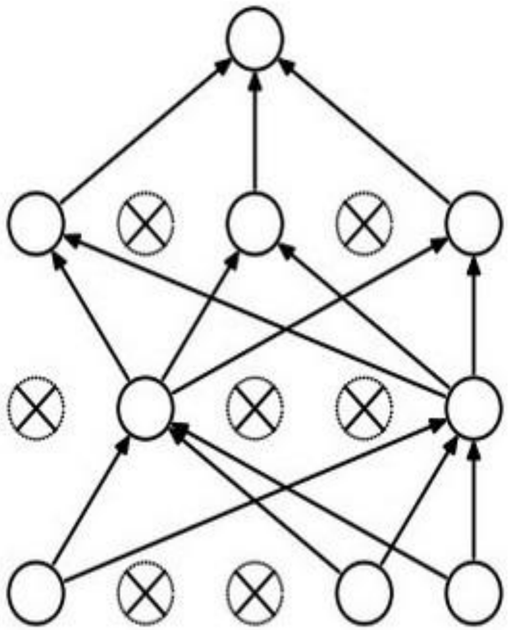
Forces the network to have a redundant representation.





http://cs231n.github.io/

How could this possibly be a good idea?

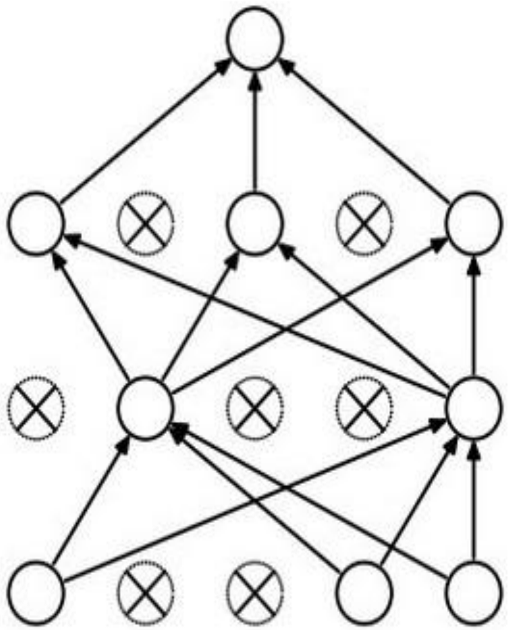


Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained with only ~one update.

At test time….



Ideally:
want to integrate out all the noise

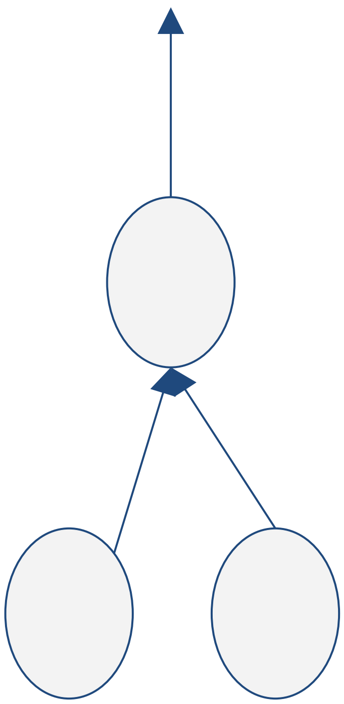
Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

http://cs231n.github.io/

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



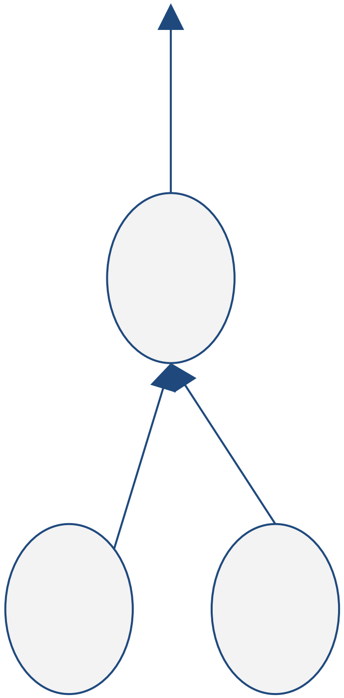
(this can be shown to be an approximation to evaluating the whole ensemble)

http://cs231n.github.io/

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



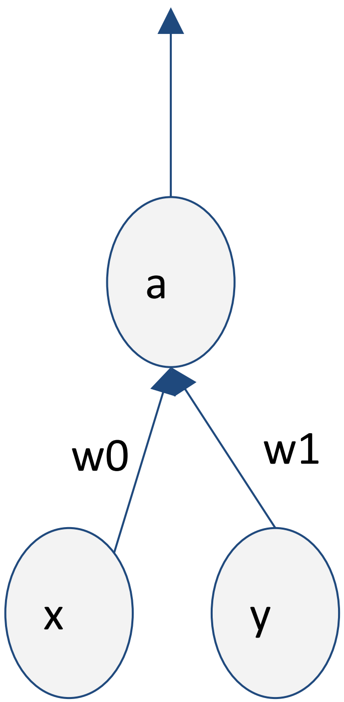
Q: Suppose that with all inputs present at test time the activation of this neuron is x.

What would its activation be during training time, in expectation? (e.g. if $p = 0.5$)

At test time....

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



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 * (2w0*x + 2w1*y)
     = ½ * (w0*x + w1*y)
```

=> Have to compensate by scaling the activations back down by ½

http://cs231n.github.io/

Dropout in Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Dropout

- Dropout can be seen as ensemble averaging
- Each model in the ensemble is smaller than the original
- Reduces overfitting
- Introduces train and test mode

- Common settings are:
  - 0.2 dropout on input layer
  - 0.5 dropout on hidden layers

Good Old L1/L2 Regularization

- Keras layer parameters

```
keras.layers.core.Dense(units, activation=None,
kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None,
bias_constraint=None)
```

- **activation**: Default is linear!
- **regularizer**: This is our standard L1 or L2 option
- Example: https://keras.io/regularizers/

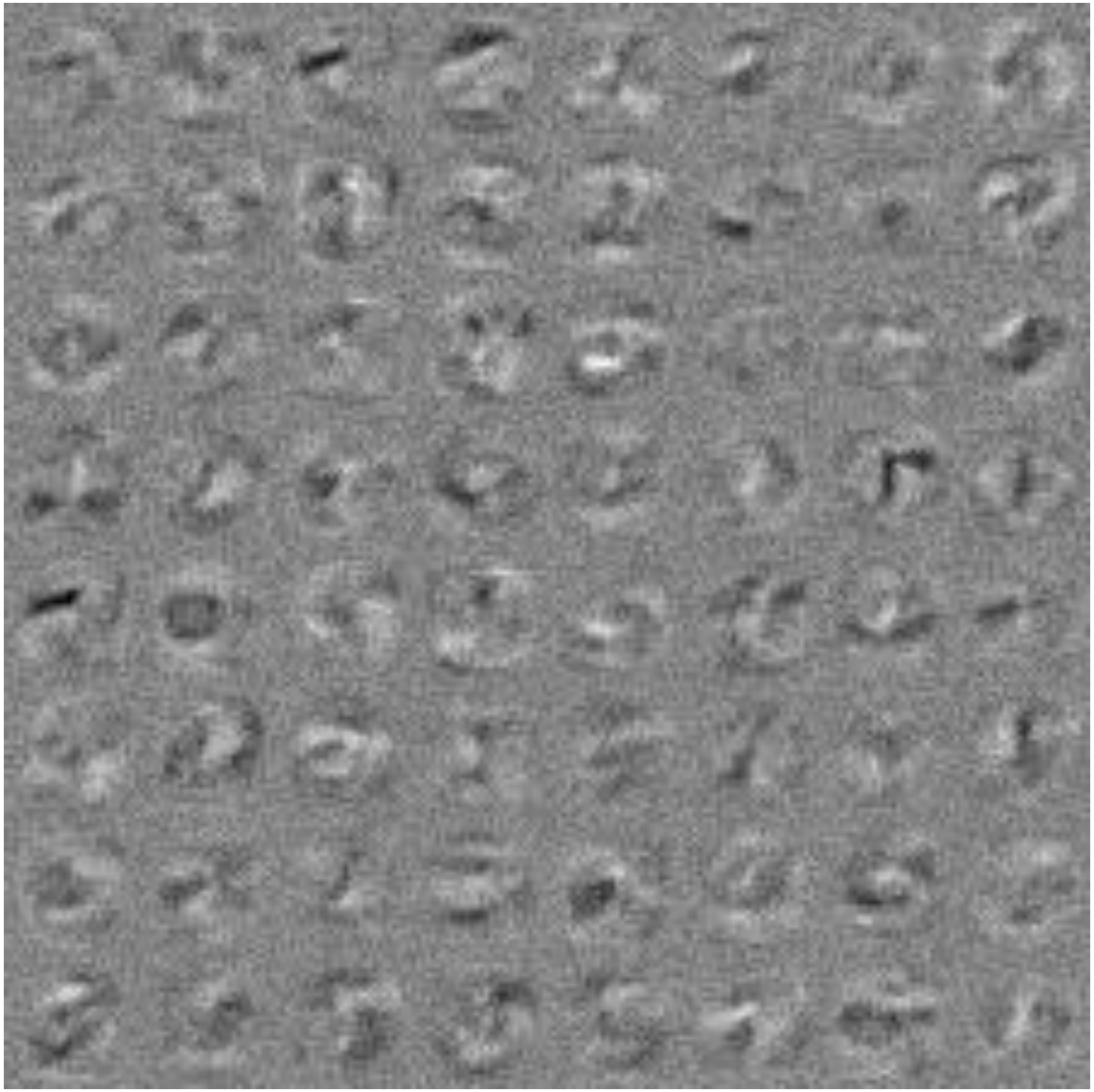
```
Dense(10, activation='relu,
      kernel_regularizer=regularizers.l2(0.01))
```

Filter visualizations

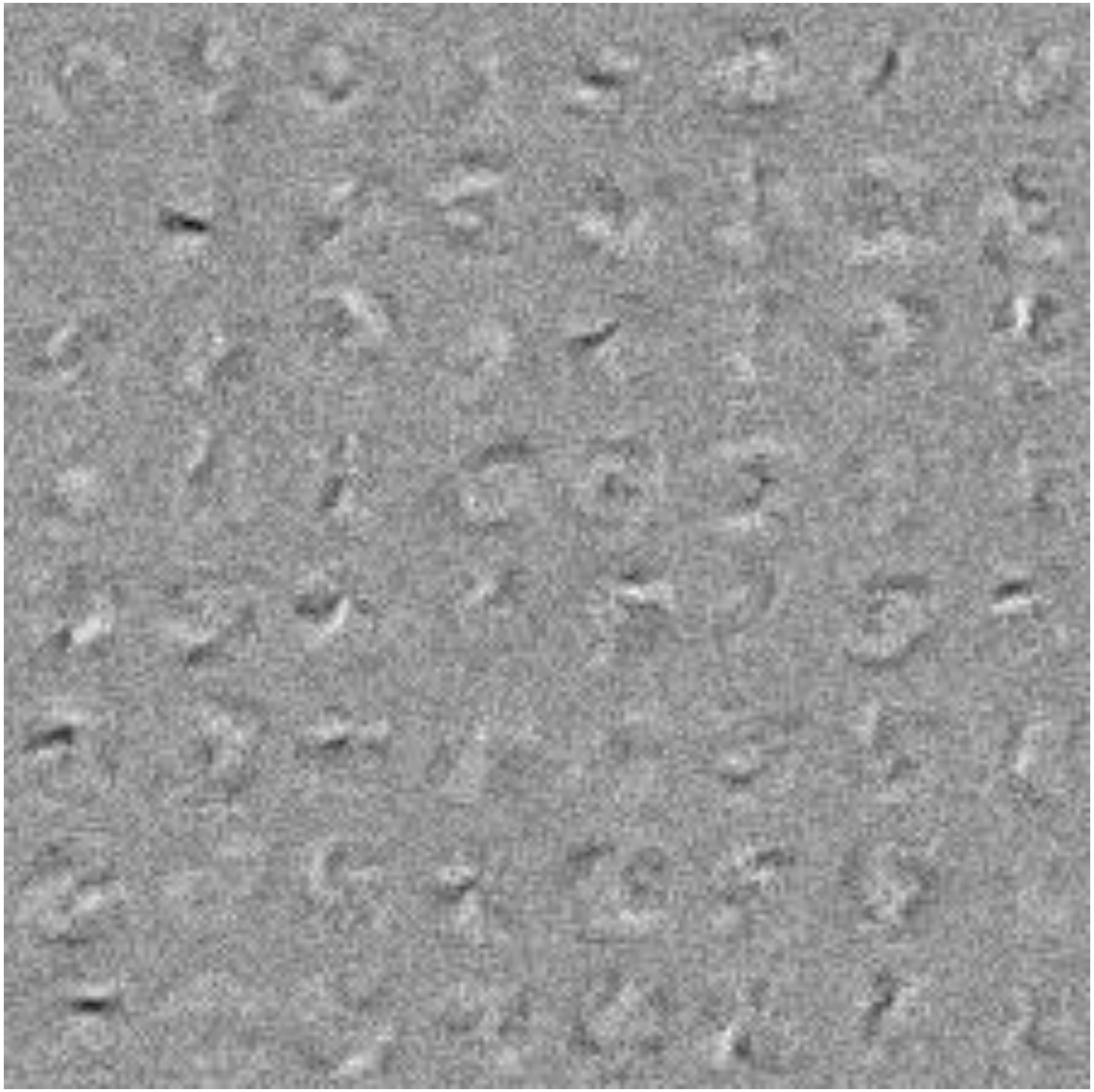
- original simple MLP network
- with regularization
- with dropout

- snapshots at 1, 10, 100, 200 epochs

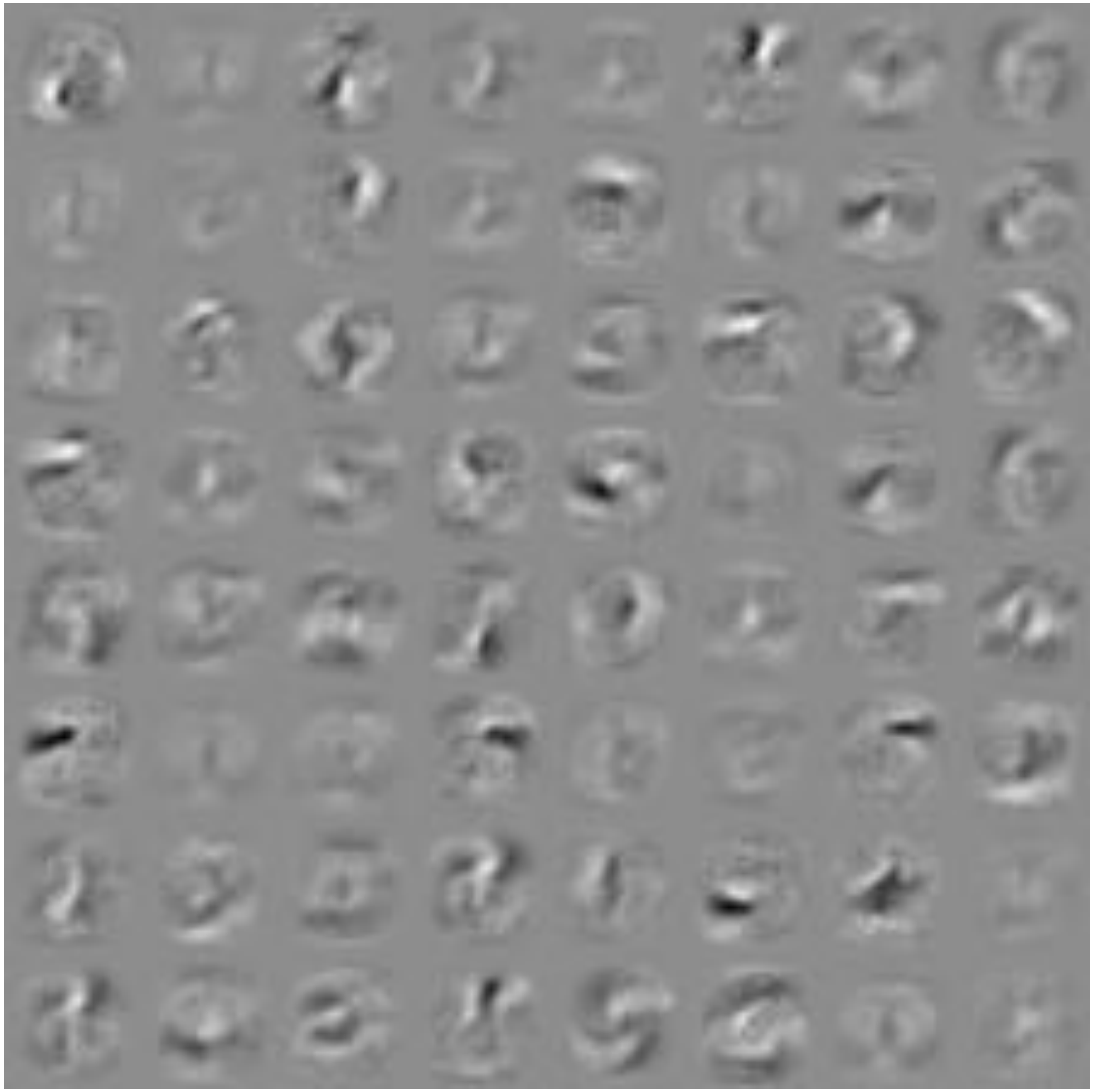
MLP with Sigmoid



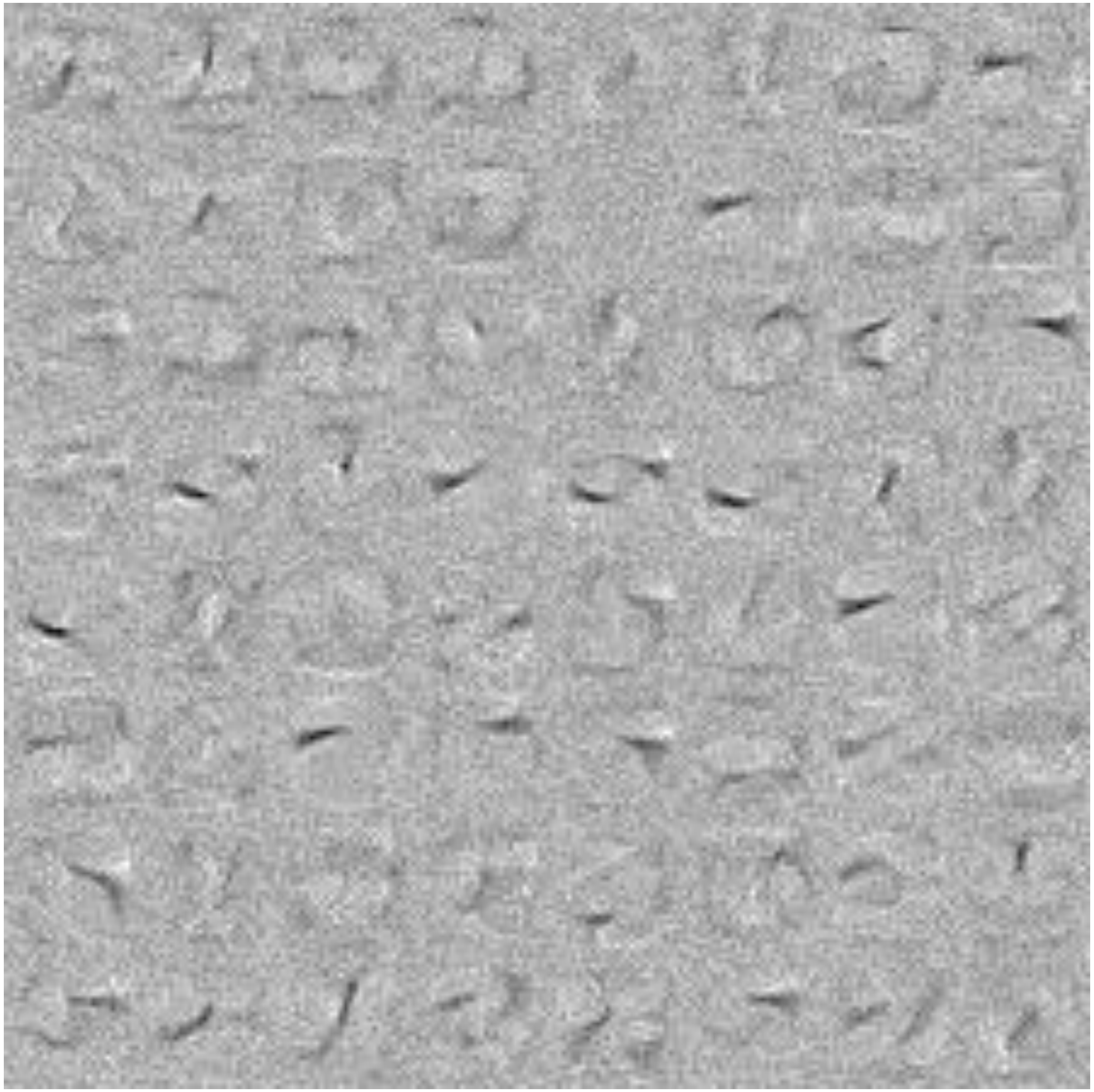
MLP with ReLU



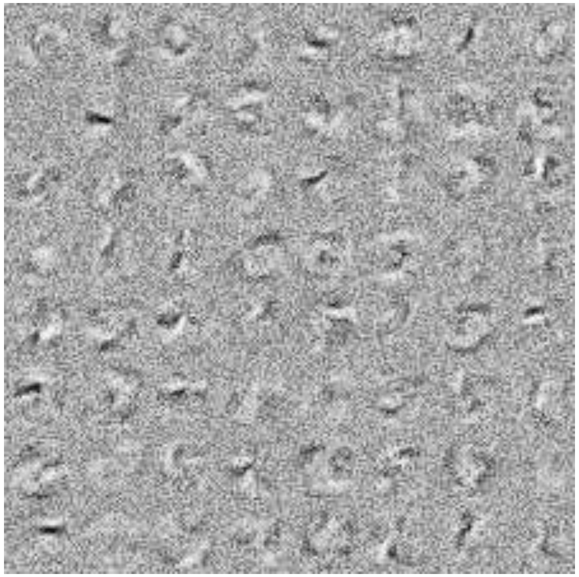
MLP ReLU + L2



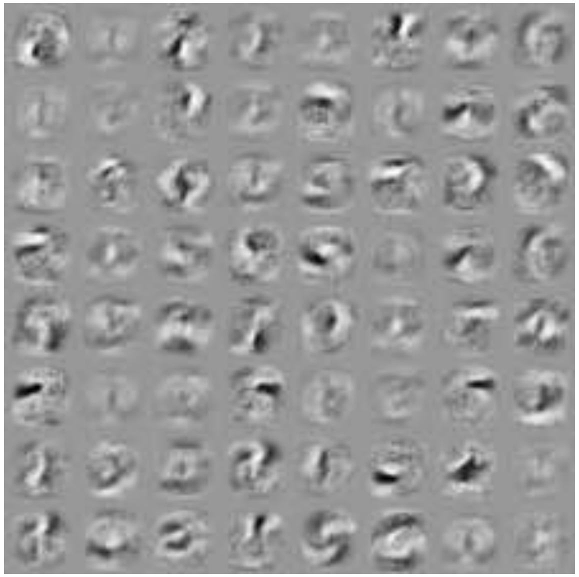
MLP ReLu + Dropout



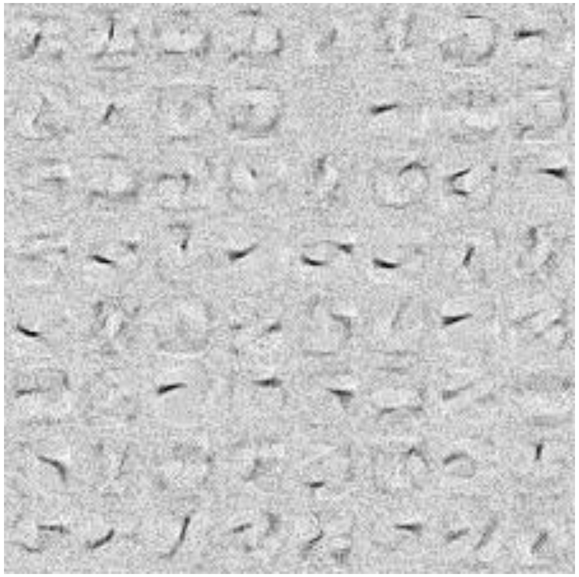
Overview



ReLU



ReLu + L2



ReLu + Dropout

Try at home: ReLU + L2 + Dropout

What About Accuracy?

200 epochs, lr = 0.001, batch_size = 100

MLP_ReLU
Test loss: 0.0887242713874
Test accuracy: 0.9757

MLP_ReLU + L2
Test loss: 0.266304204607
Test accuracy: 0.968

MLP_ReLU + Dropout
Test loss: 0.135238234081
Test accuracy: 0.9638

This is not a fair comparison!

It is typical for dropout to increase training time

Milestone!

- What we have so far:
  - CNN layer
  - MaxPool layer
  - FC layer
  - Activations:
    - **ReLU**
    - sigmoid or softmax for last layer
  - Cost function – cross entropy
  - **Training – SGD + Momentum**
  - Regularization – dropout + L2/L1
  - Convergence check

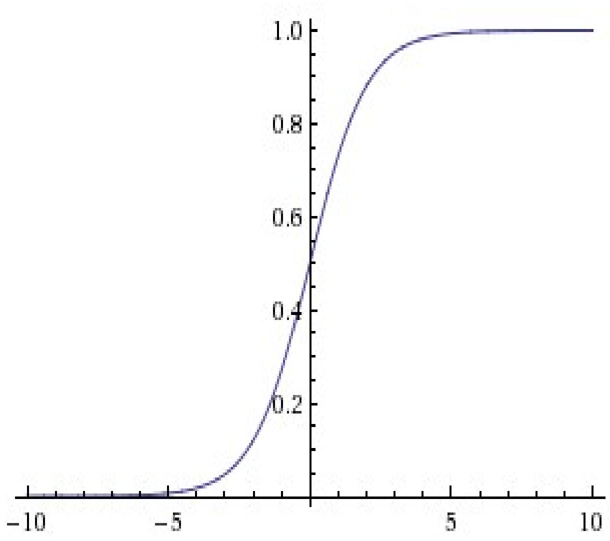
A few more notes on ReLU and Gradients

- One of the tricks that made deep learning work
- Need to be a bit careful

- We train our network with gradient information
- We take the gradient of our loss function
- And find the parameters that correspond to a good local minimum

Activation Functions

$$\sigma(x) = 1/(1+e^{-x})$$



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

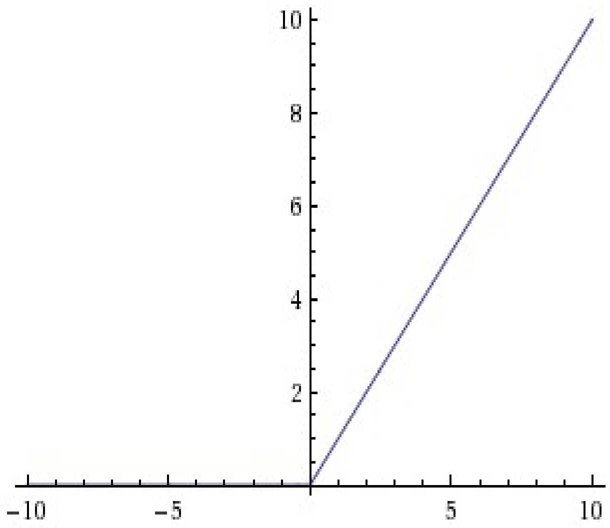
problems:

1. Saturated neurons "kill" the gradients
2. exp() is a bit compute expensive

$$\boxed{\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}}$$

http://cs231n.github.io/

Activation Functions



ReLU
(Rectified Linear Unit)

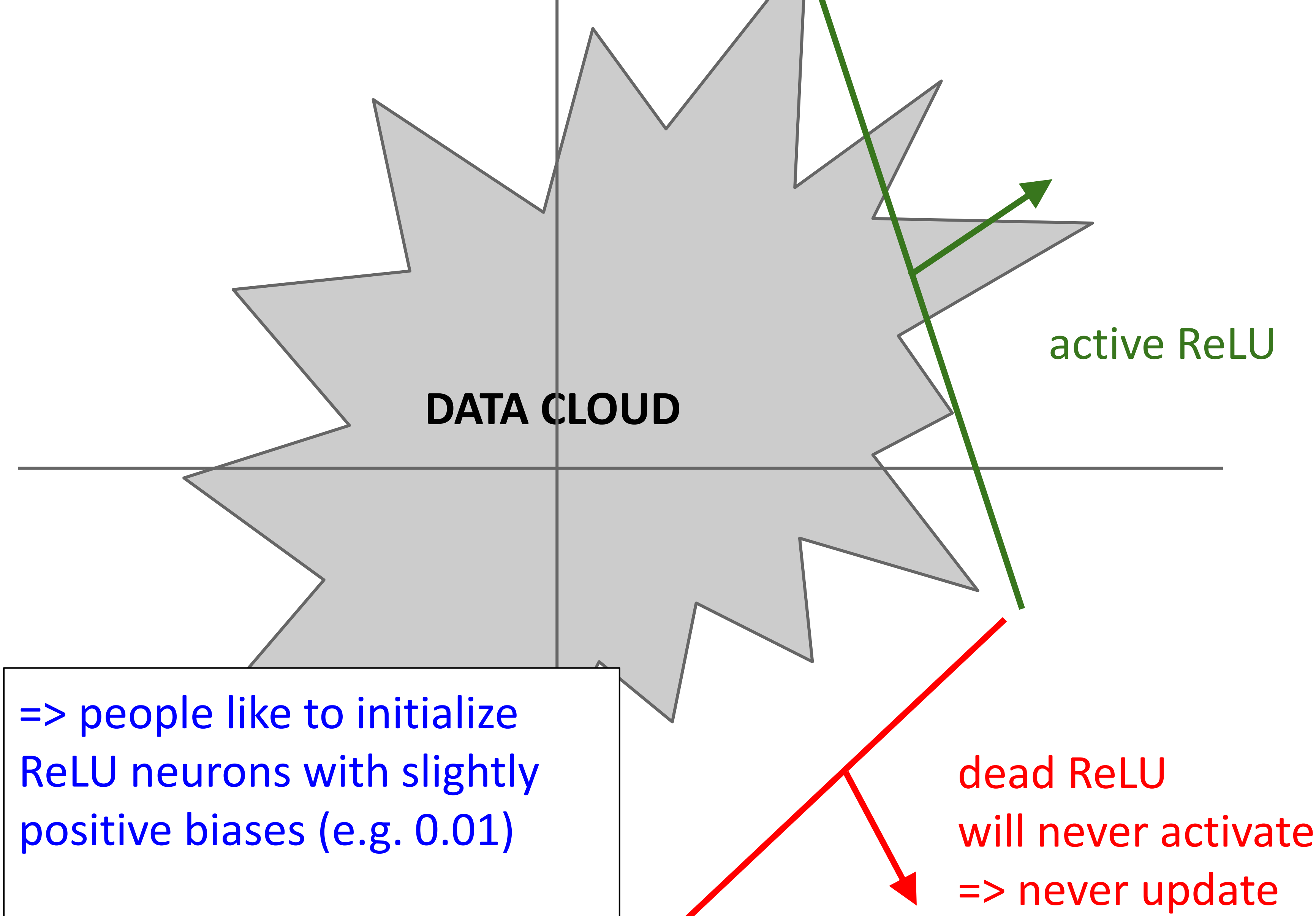
- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

http://cs231n.github.io/



=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

http://cs231n.github.io/

ReLU in Practice

- ReLU is a good default to try
- Be careful with your learning rate
- Might benefit from small positive bias initialization

Weight Initialization

- Typically random if you train from scratch
- If weights are too large the network has a hard time learning from updates
- If weights are too small they might not break symmetry enough.

- This is actually an area of ongoing research.

Proper initialization is an active area of research…

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

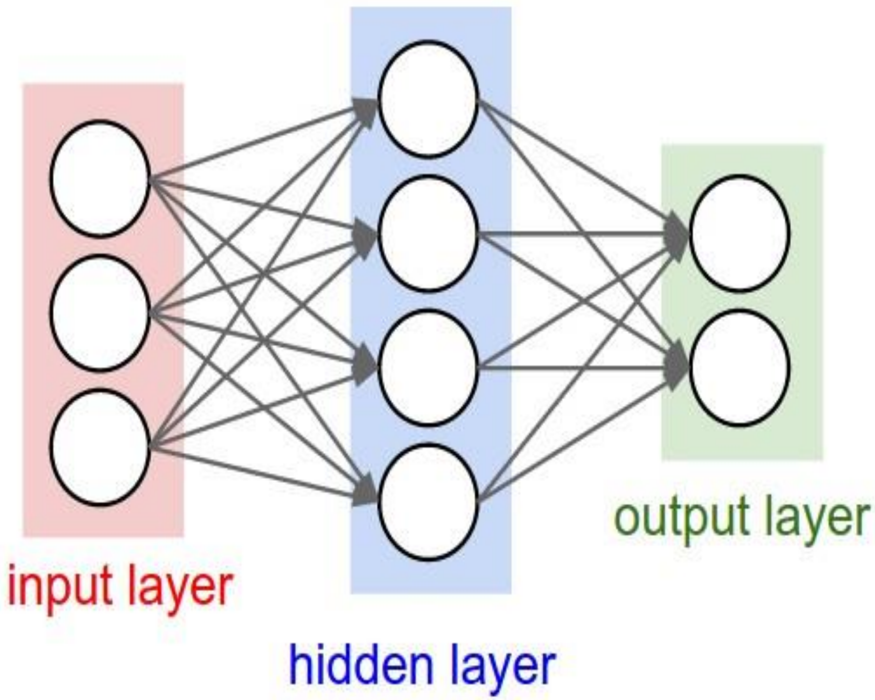
Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015
…

- Q: what happens when W=0 init is used?



http://cs231n.github.io/

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Weight Initialization Confusion

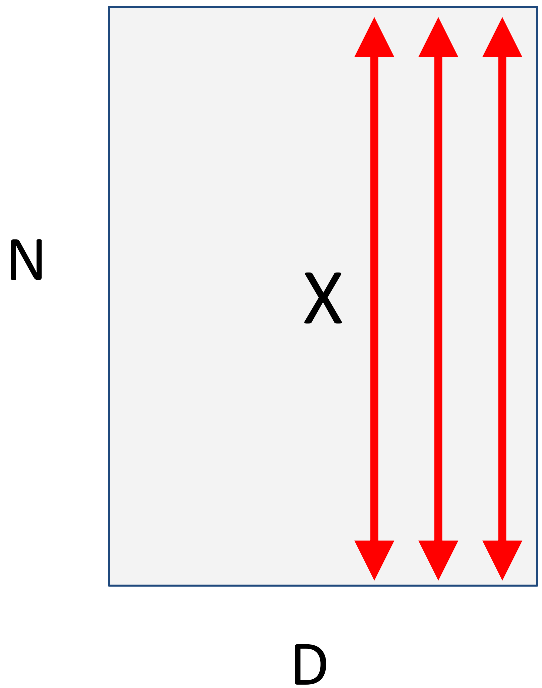
- glorot (also called Xavier)(for tanh)
  - stddev = sqrt(2 / (fan_in + fan_out)) [Keras]
  - OR
  - stddev = sqrt(1 / fan_in) [Caffe]

- he (for ReLU)
  - stddev = sqrt(2 / fan_in) [Keras]

[Ioffe and Szegedy, 2015]

The Rescue: BatchNorm Layer

"you want unit gaussian activations?
just make them so."



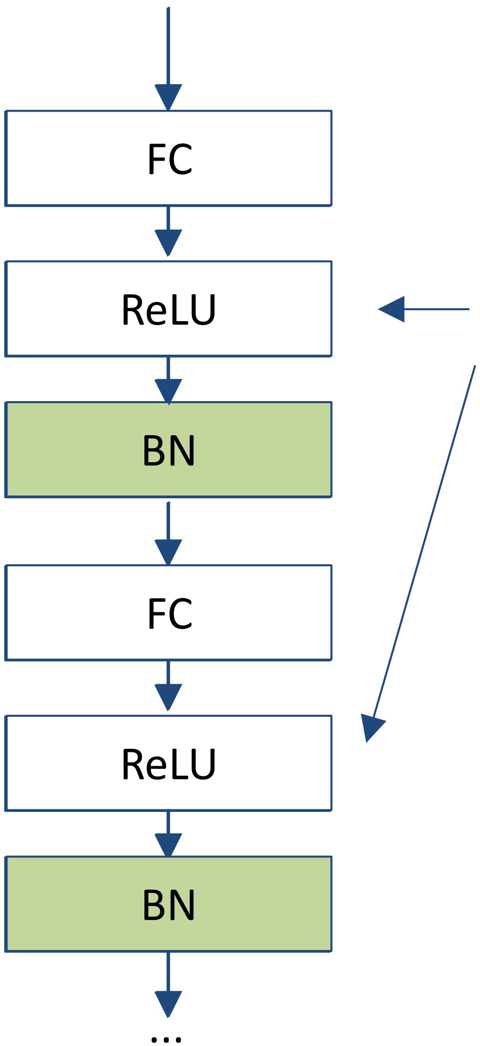
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after fully connected or convolutional layers, and before nonlinearity....
Or After....

- Improves gradient flow through the network
- Allows higher learning rates
- **Reduces the strong dependence on initialization**
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Better Optimization

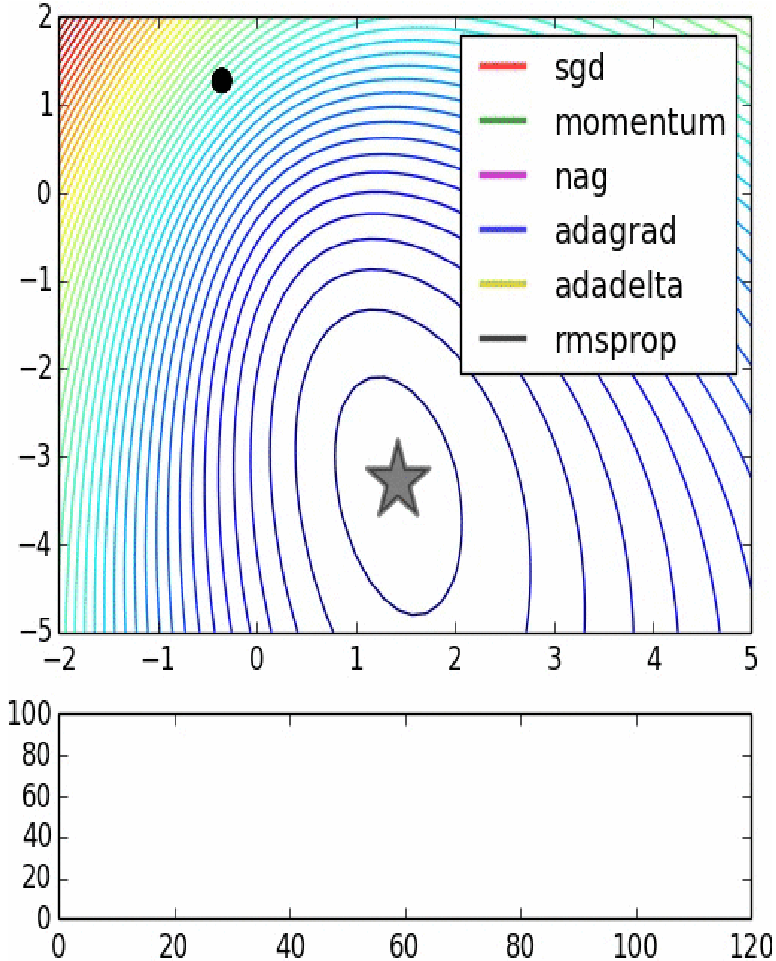


Image credits: Alec Radford

http://cs231n.github.io/

AdaGrad update

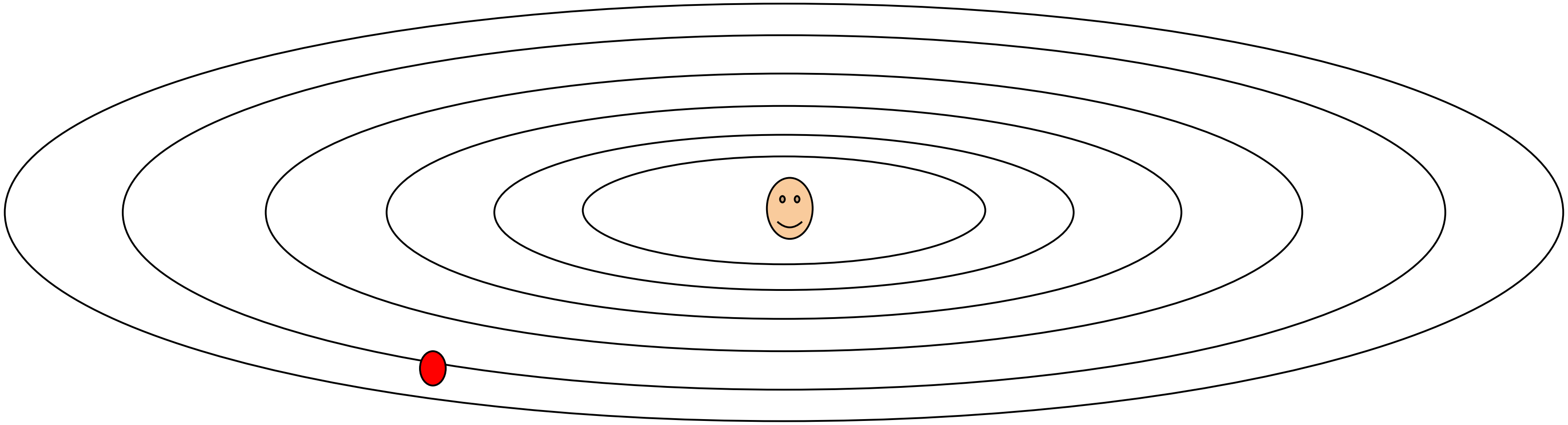
[Duchi et al., 2011]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension



Over time step size gets very small – training comes to a halt.

http://cs231n.github.io/

RMSProp update
Decay the cache over time

[Tieleman and Hinton, 2012]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

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

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

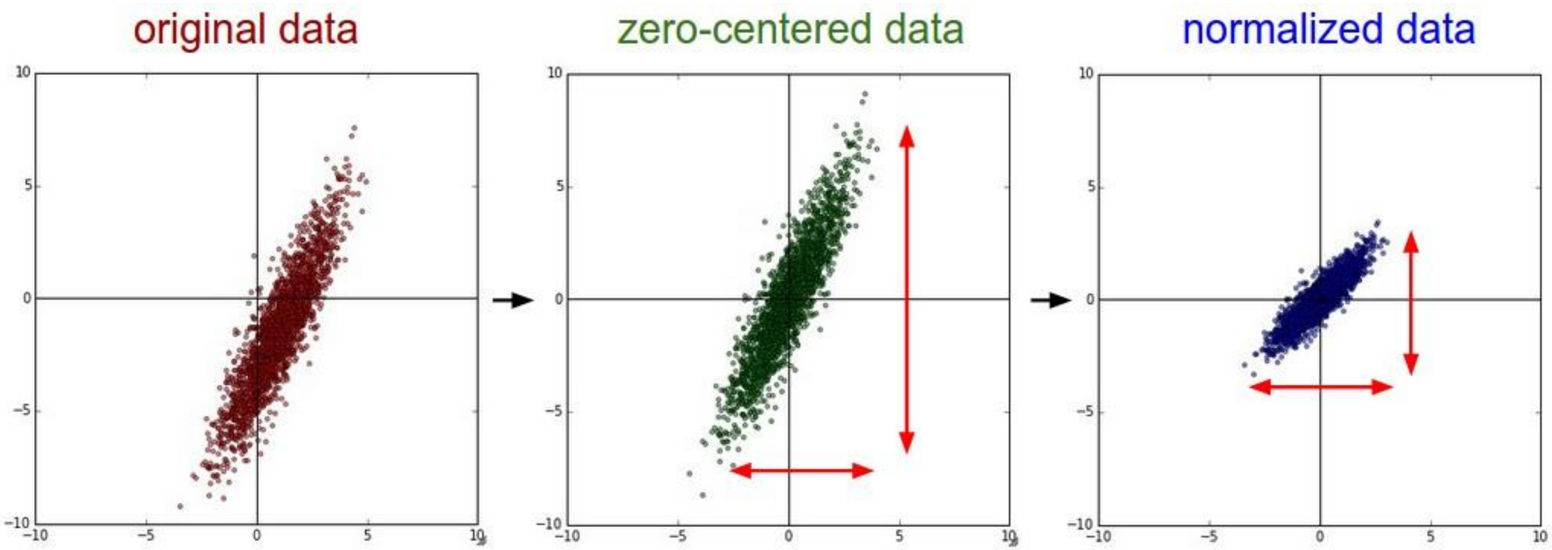
Looks a bit like RMSProp with momentum

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

Milestone Update

- What we have so far:
  - CNN layer
  - MaxPool layer
  - FC layer
  - **BatchNorm Layer**
  - Activations:
    - **ReLU**
    - sigmoid or softmax for last layer
  - Cost function – cross entropy
  - **Initialization**
  - **Training – Adam**
  - Regularization – dropout + L2/L1
  - Convergence check

Preprocess the data



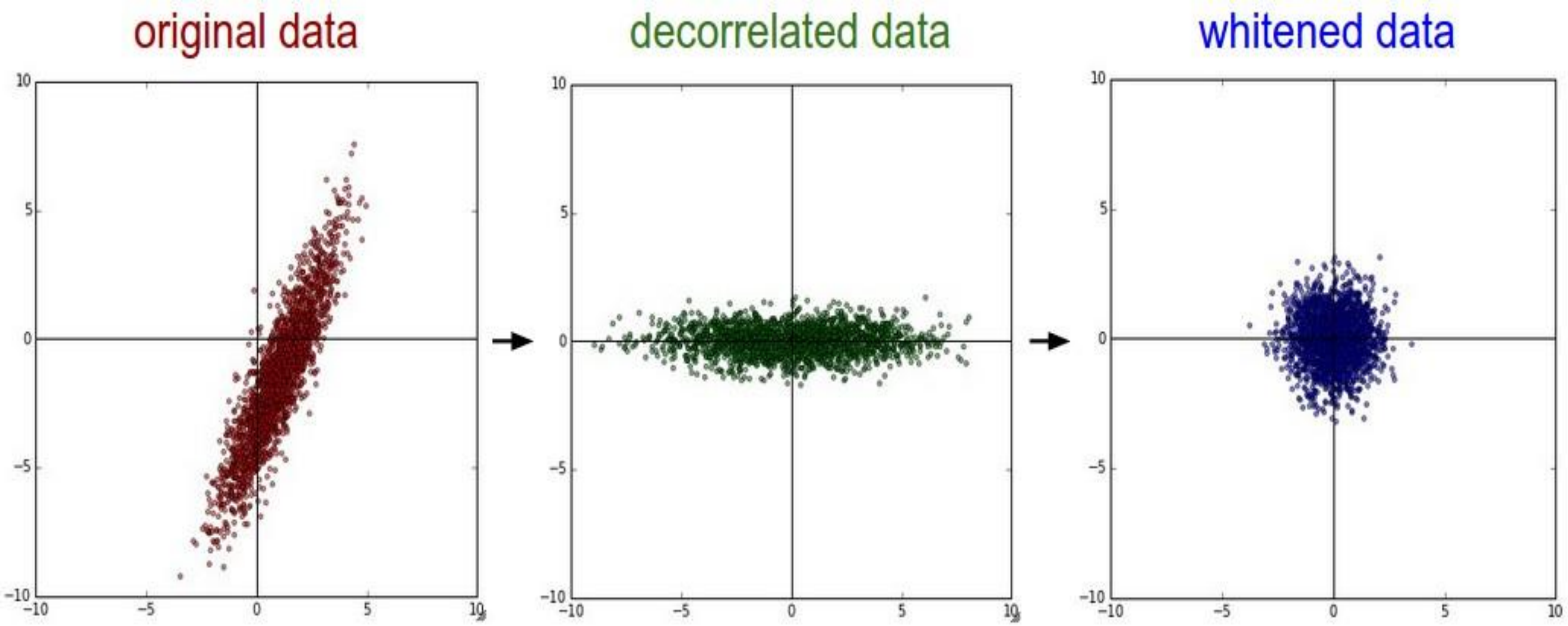

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

http://cs231n.github.io/

Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

http://cs231n.github.io/

TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening

http://cs231n.github.io/

Data Preprocessing in Keras

```
keras.preprocessing.image.ImageDataGenerator(
      featurewise_center=False,
      samplewise_center=False,
      featurewise_std_normalization=False,
      samplewise_std_normalization=False,
      zca_whitening=False,
      …
      )
```

https://keras.io/preprocessing/image/

Remember

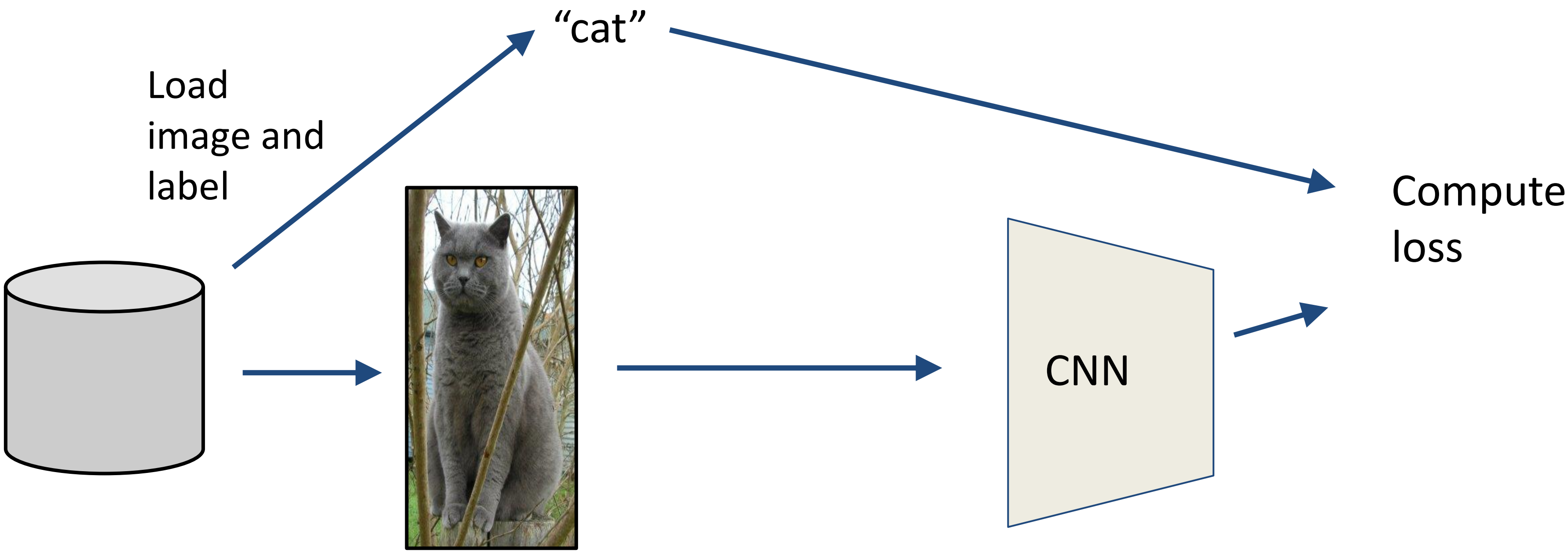
- You need to handle the test data in the same way!
- Save normalization parameters
- Apply them to test data

- In scikit learn terminology:

```
normalizer.fit_transform(train_data)
normalizer.transform(test_data)
```

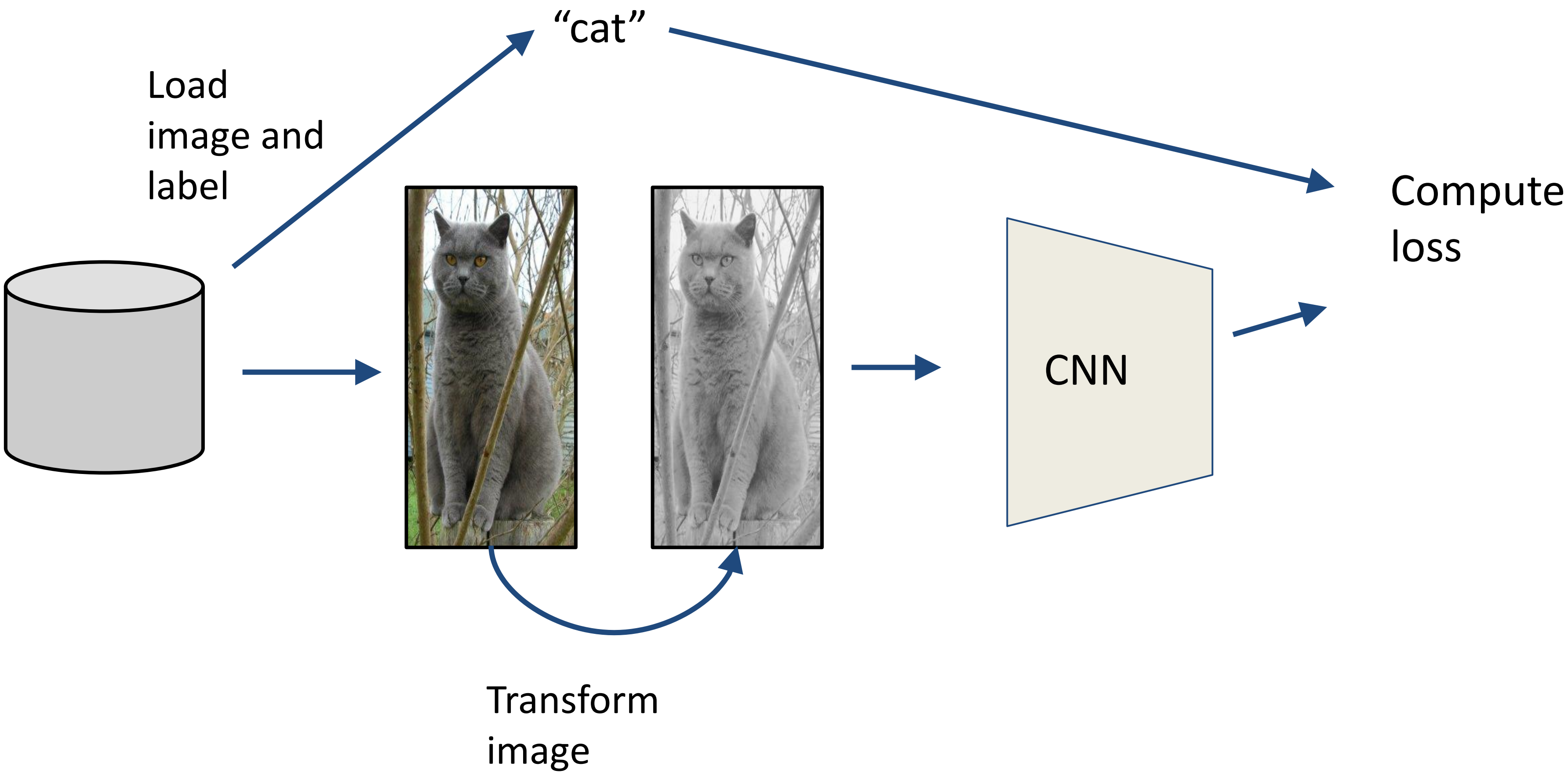
Data Augmentation

Data Augmentation



http://cs231n.github.io/

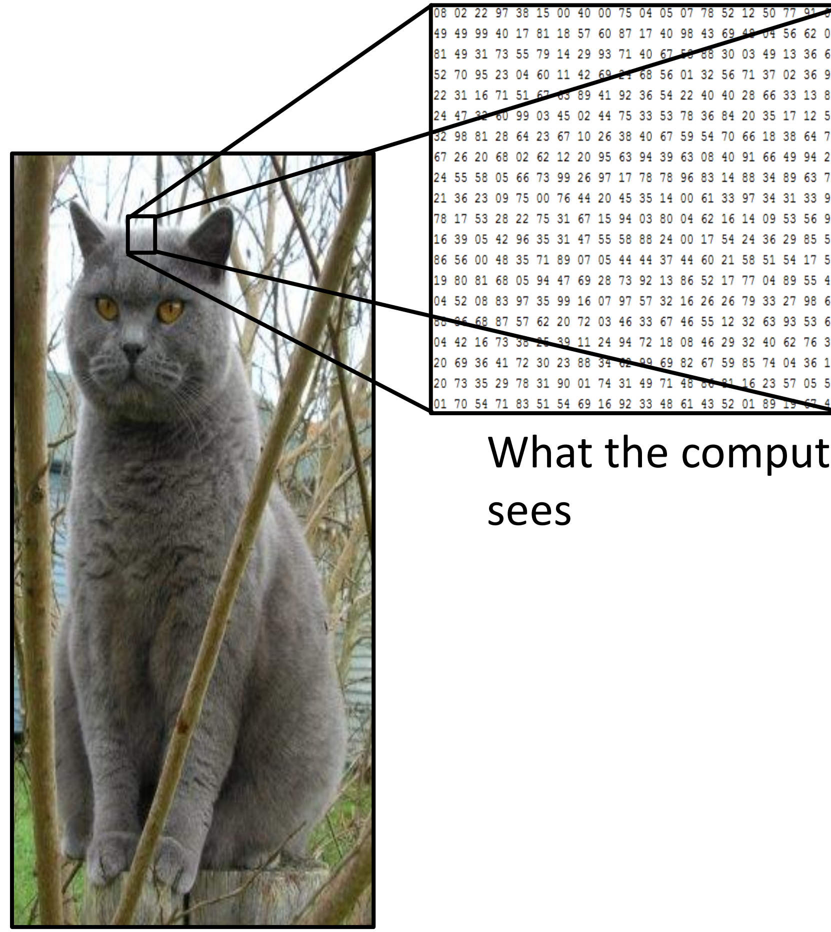
Data Augmentation



http://cs231n.github.io/

Data Augmentation

- Change the pixels without changing the label
- Train on transformed data
- VERY widely used

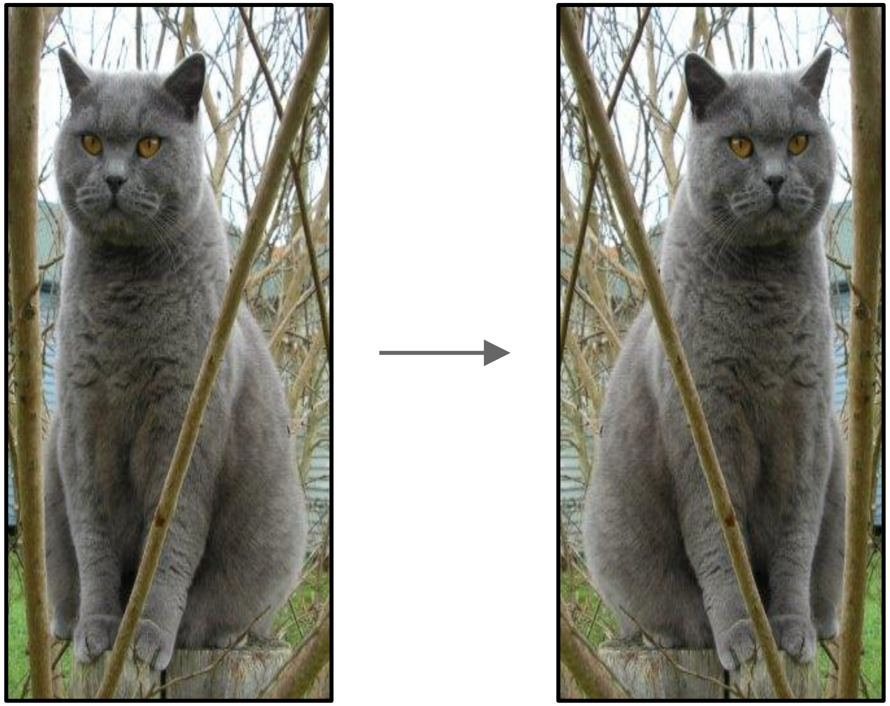


What the computer sees

http://cs231n.github.io/

Data Augmentation

1. Horizontal flips

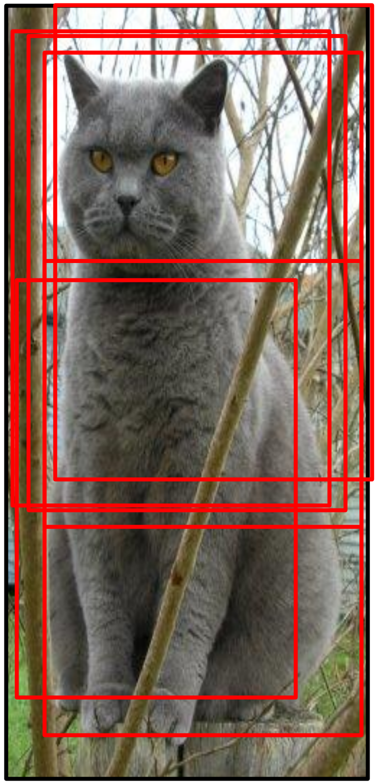


http://cs231n.github.io/

Data Augmentation

2. Random crops/scales

Training: sample random crops / scales



http://cs231n.github.io/

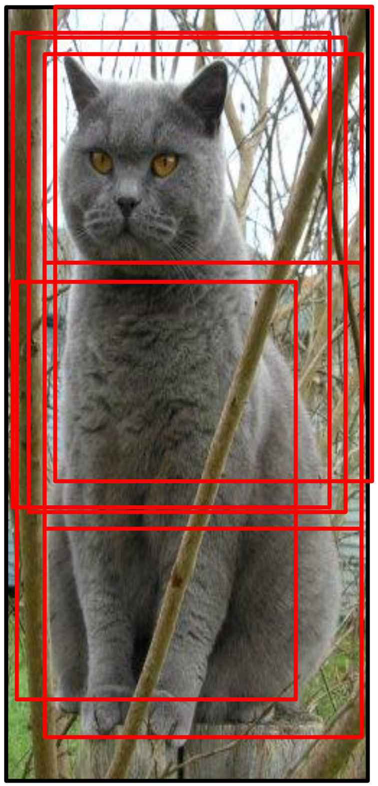
Data Augmentation

2. Random crops/scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



http://cs231n.github.io/

Data Augmentation

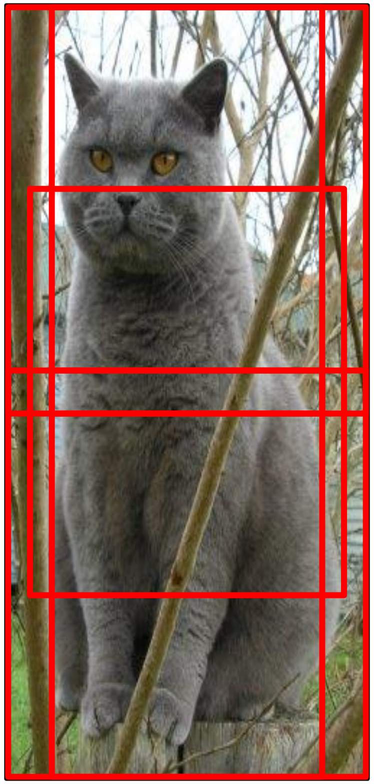
2. Random crops/scales

Training: sample random crops / scales

ResNet:

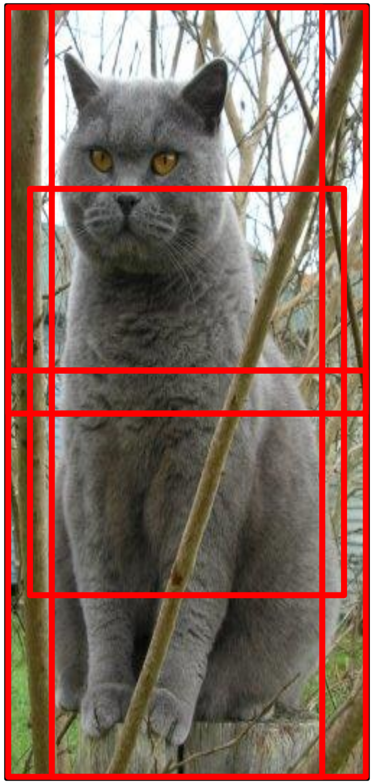
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average a fixed set of crops



Data Augmentation

2. Random crops/scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

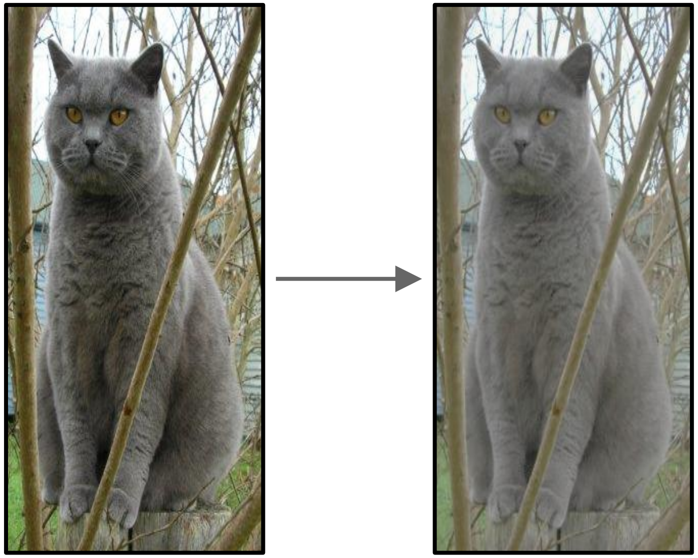
Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

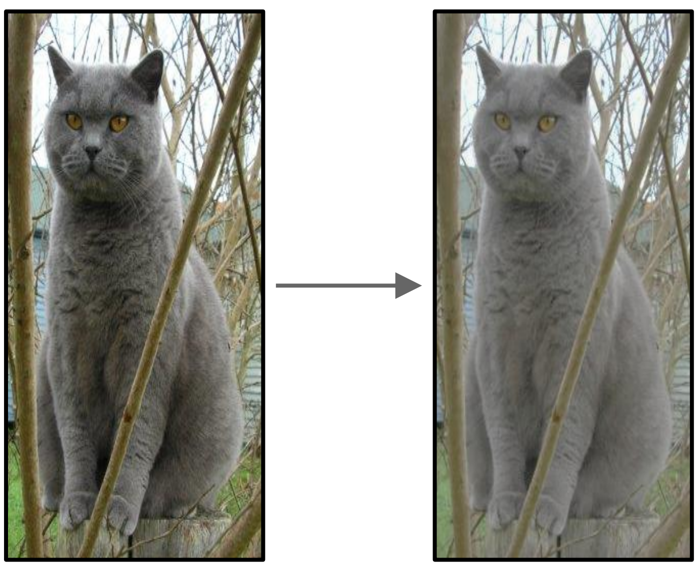
Data Augmentation
3. Color jitter

Simple:
Randomly jitter contrast



Data Augmentation
3. Color jitter

Simple:
Randomly jitter contrast



Complex:

1. Apply PCA to all [R, G, B] pixels in training set

2. Sample a “color offset” along principal component directions

1. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012],* ResNet, etc)

Data Augmentation

4. Get creative!

Random mix/combinations of :
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Data Augmentation: Takeaway

- Simple to implement, use it
- Especially useful for small datasets
- Fits into framework of noise / marginalization

Data Augmentation in Keras

```
keras.preprocessing.image.ImageDataGenerator(
      …
      rotation_range=0., width_shift_range=0.,
      height_shift_range=0., shear_range=0.,
      zoom_range=0., channel_shift_range=0.,
      fill_mode='nearest', cval=0.,
      horizontal_flip=False, vertical_flip=False,
      rescale=None, preprocessing_function=None,
      data_format=K.image_data_format())
      )
```

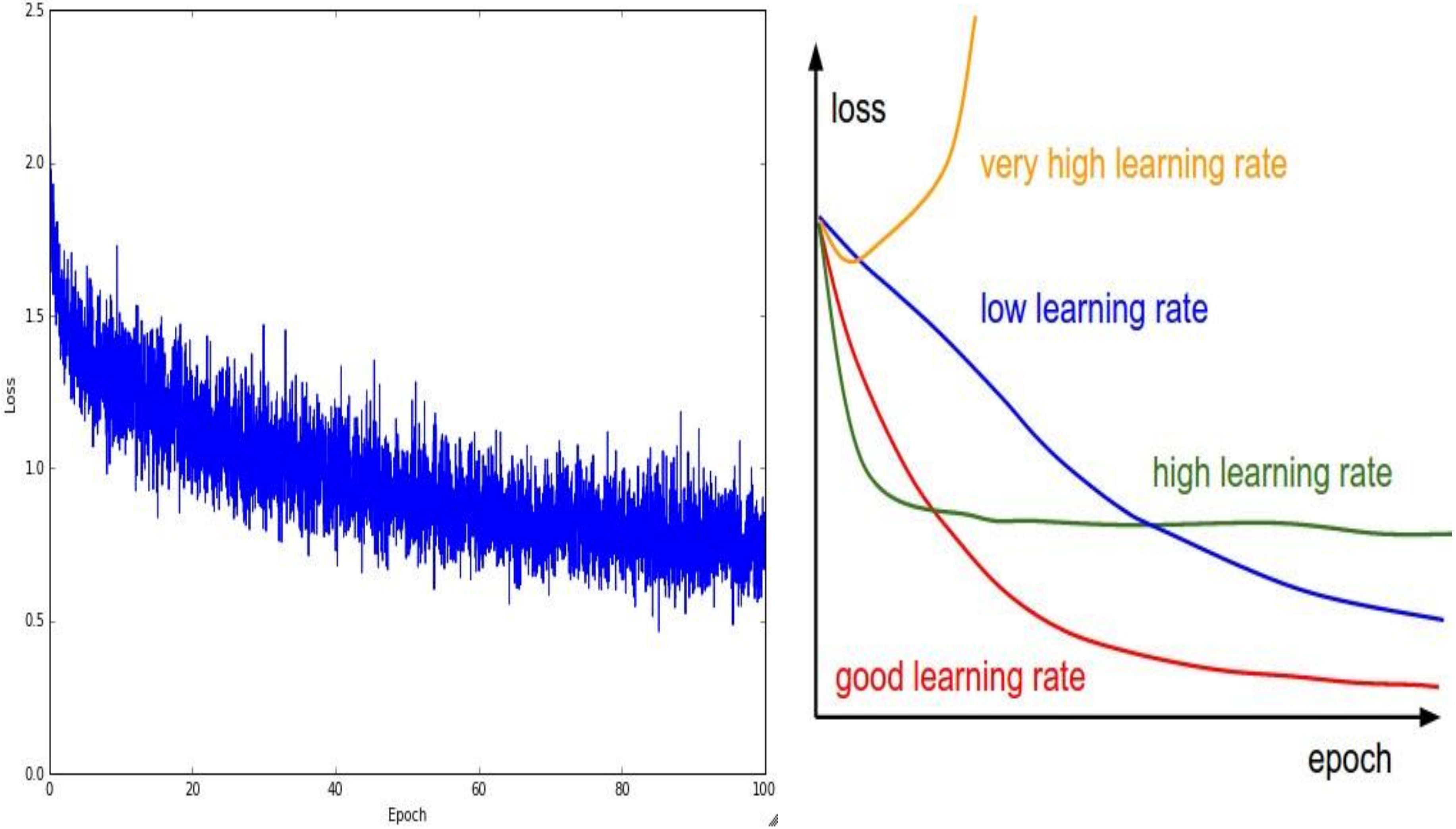
https://keras.io/preprocessing/image/

Learning Rate (once again)

- Need to find good range for learning rate
  - Look at the loss during training
  - If it is NaN => learning rate too high
  - If it decreases too slow => learning rate too low

- Also remember batch size and gradient estimation influence
  - Large batch – large learning rate – fewer updates
  - Small batch – smaller learning rate – more updates

Monitor and visualize the loss curve

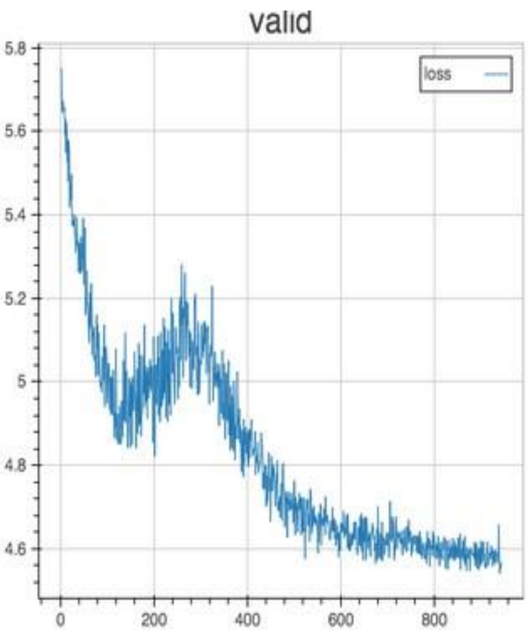


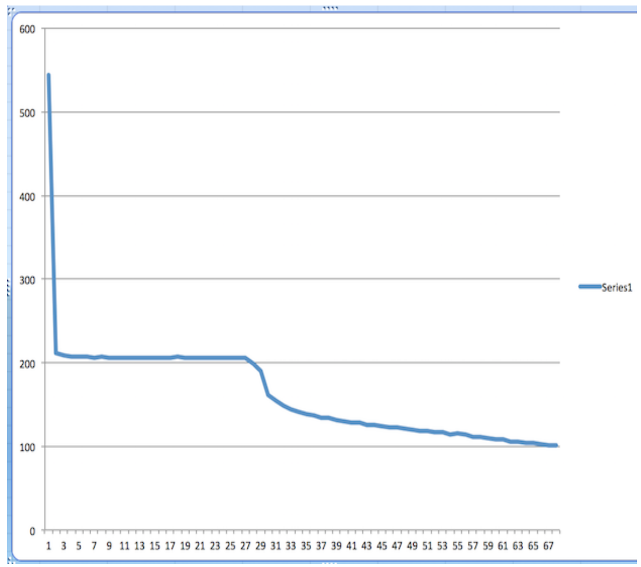
http://cs231n.github.io/

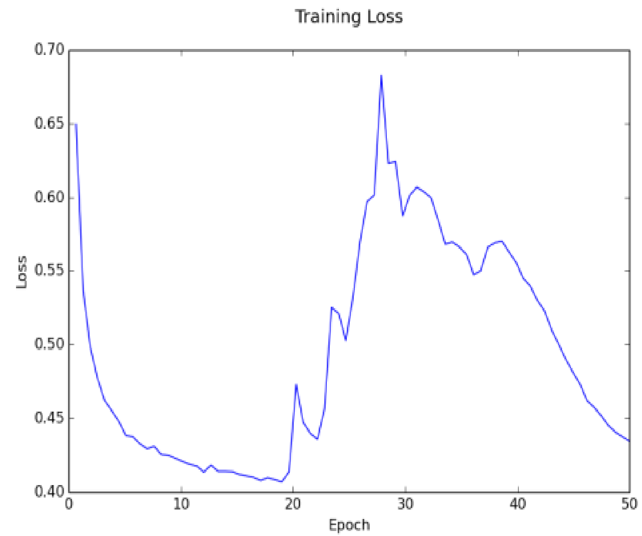
lossfunctions.tumblr.com

Loss function specimen

Andrey Karpathy



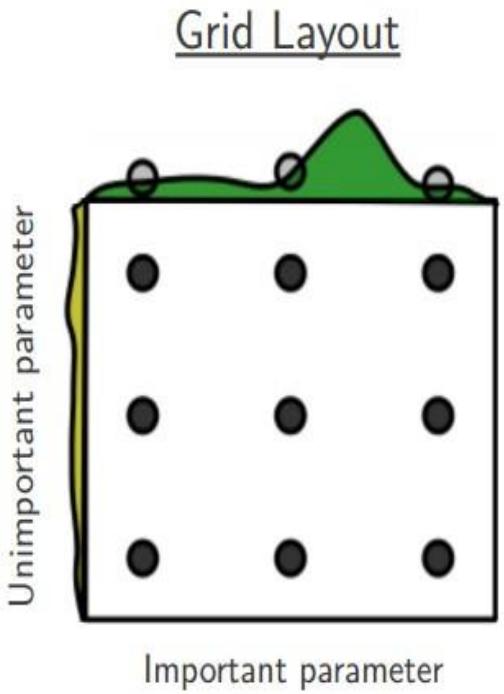


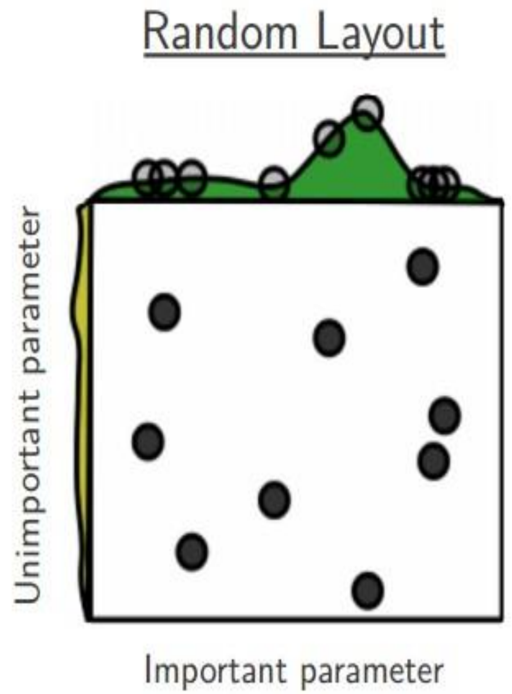


Cross Validation for Parameter Tuning

- How to fine tune the learning rate and regularization?
- Just like any classifier you can also use CV for deep learning
- Costs for trying a parameter setting are high
- People tend to use coarse grids
- This can be bad

Random Search vs. Grid Search





Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Architecture Design

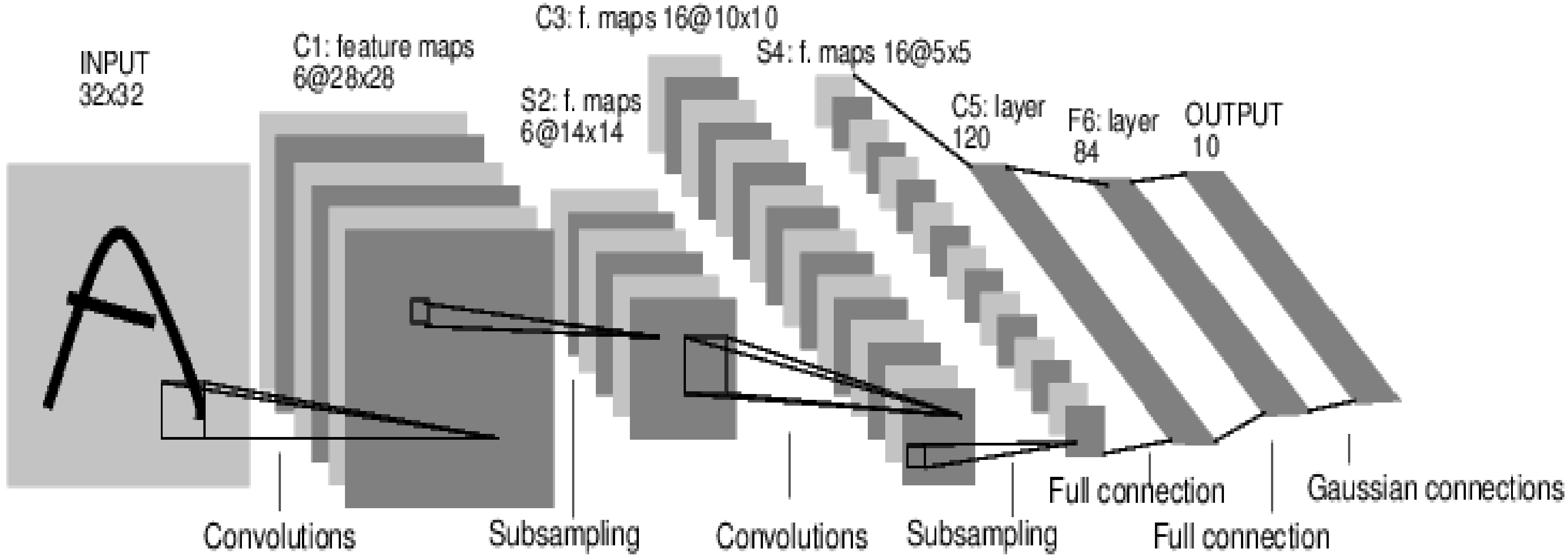
- Things to consider:
  - Previous work done on similar problems
  - Computational resources
- Wider, and deeper means more degrees of freedom
  - Better classification performance
  - IF you can tune it right

Case studies

- Don‘t go from scratch when you learn deep learning
- Look at example models
- Use tricks that have been shown to work

Case Study: LeNet-5

[LeCun et al., 1998]

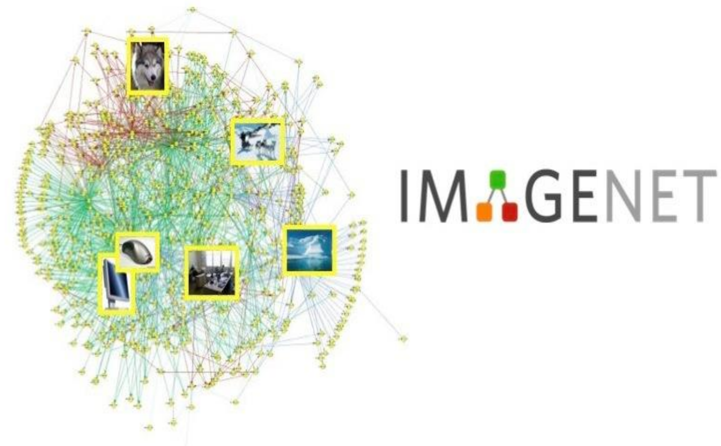


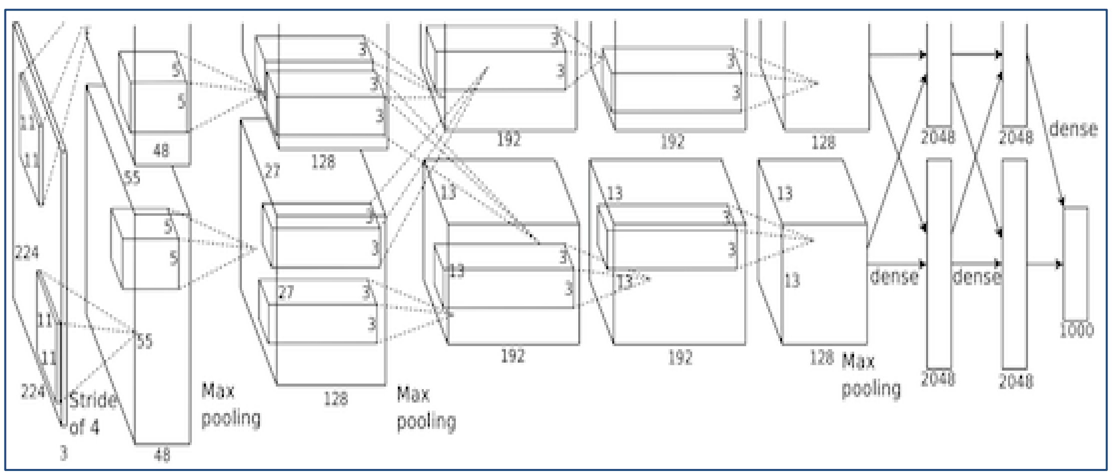
Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]



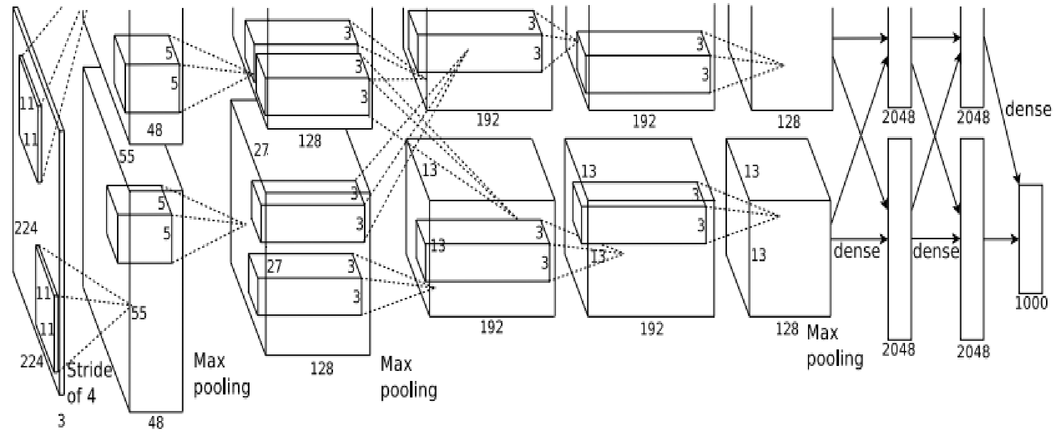


“AlexNet”

http://cs231n.github.io/

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- **7 CNN ensemble: 18.2% -> 15.4%**

http://cs231n.github.io/

Case Study: VGG-16

- Simonyan & Zisserman won the ImageNet ILSVRC-2014 challenge.
- They published a paper called “Very Deep Convolutional Networks for Large-Scale Image Recognition”
- They compare networks of up to 19 layers.
- They also **published** their network
- It is one of the most popular pre-trained networks and available for a lot of deep learning libraries, including Keras.
- https://keras.io/applications/

VGG-16 Architecture

input: 224x244 RGB image

64 Conv 3x3
64 Conv 3x3
maxpool

128 Conv 3x3
128 Conv 3x3
maxpool

256 conv 3x3
256 conv 3x3
256 conv 3x3
maxpool

512 conv 3x3
512 conv 3x3
512 conv 3x3
maxpool
512 conv 3x3
512 conv 3x3
512 conv 3x3
maxpool
4096 FC
4096 FC
1000 FC
softmax

All layers using ReLU
Trained with SGD and momentum

VGG-16 - Training

- SGD + momentum
- With Gaussian initialization training needed tricks:
  - First trained smaller network (less layers)
  - Used the smaller network as initialization for deeper network
- With Glorot initialization it could be trained from scratch!
- See paper for further details: https://arxiv.org/pdf/1409.1556.pdf