Training Neural Networks

Jaegul Choo (주재걸)
Korea University
https://sites.google.com/site/jaegulchoo/

Contents

1. Activation functions
   - Sigmoid, ReLU, ...

2. Batch normalization

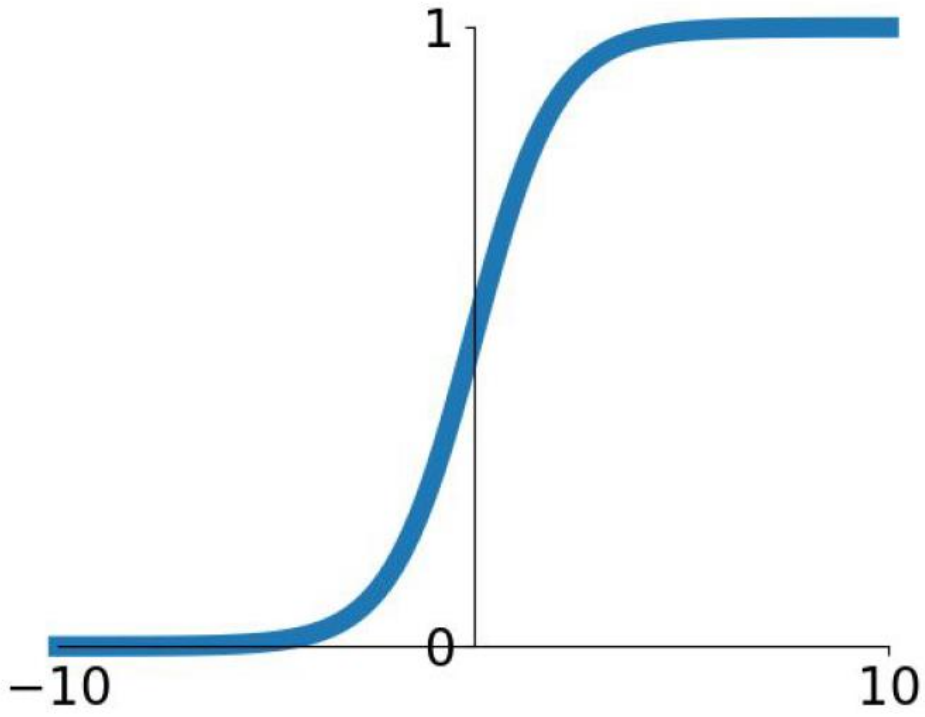
3. Optimization methods
   - SGD, momentum, Adagrad, RMSProp, Adam, AdaGrad

4. Ensemble and regularization
   - Dropout, data augmentation, ...

Activation Functions

Activation Functions

Sigmoid activation



Sigmoid

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

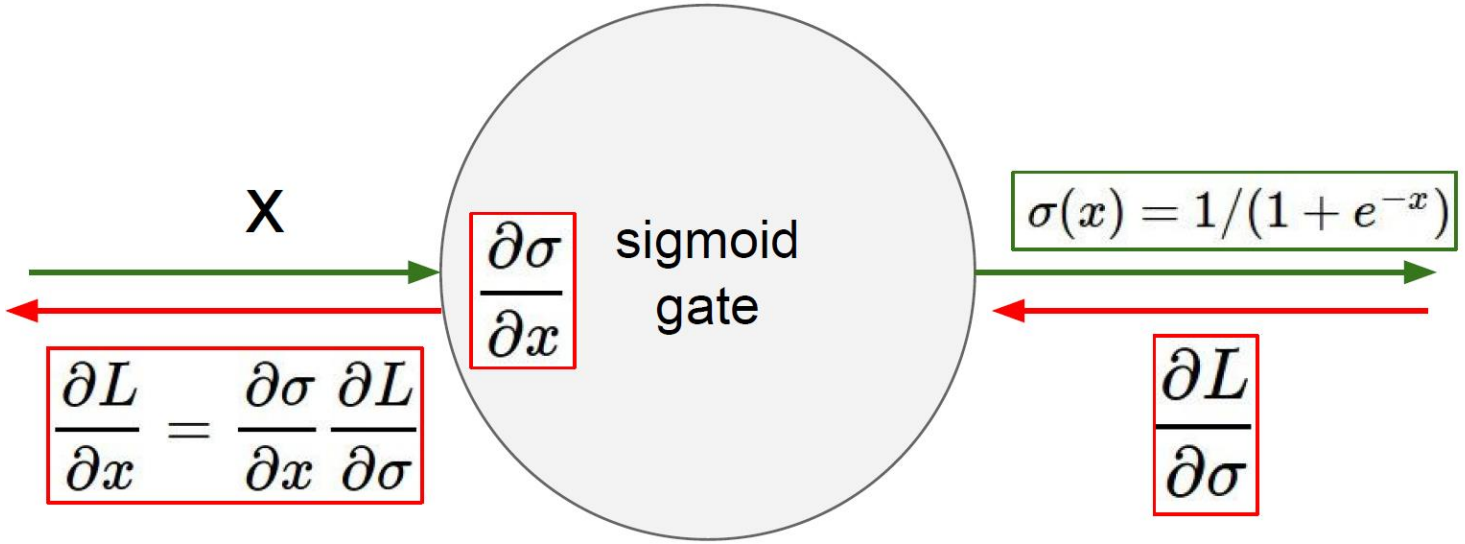
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

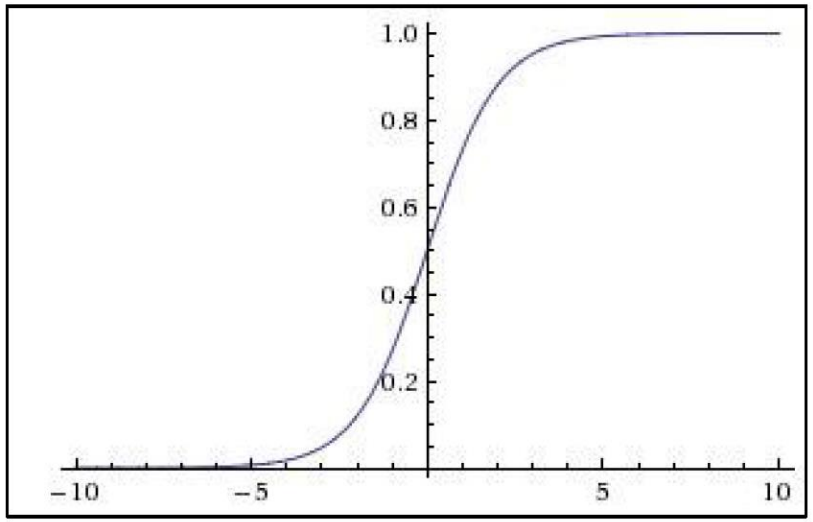
3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions

Sigmoid activation

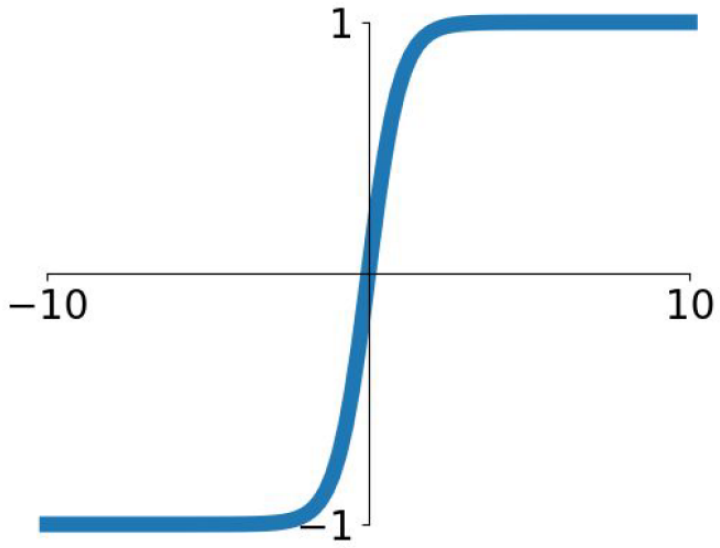




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Activation Functions

tanh activation

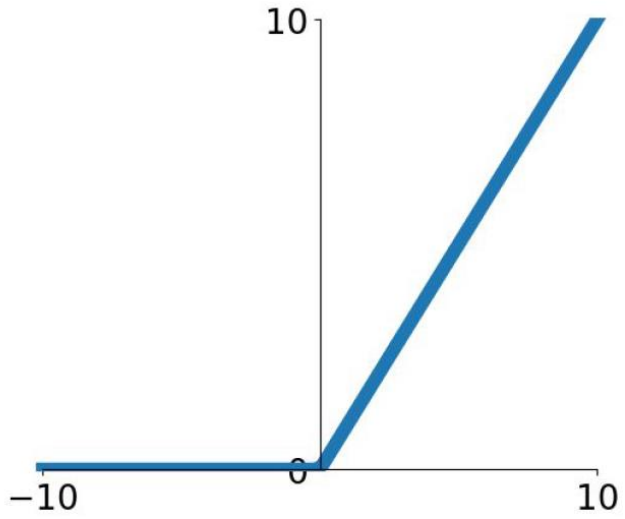


tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Activation Functions

ReLU (Rectifier Linear Unit)



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)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Batch Normalization

Batch Normalization

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

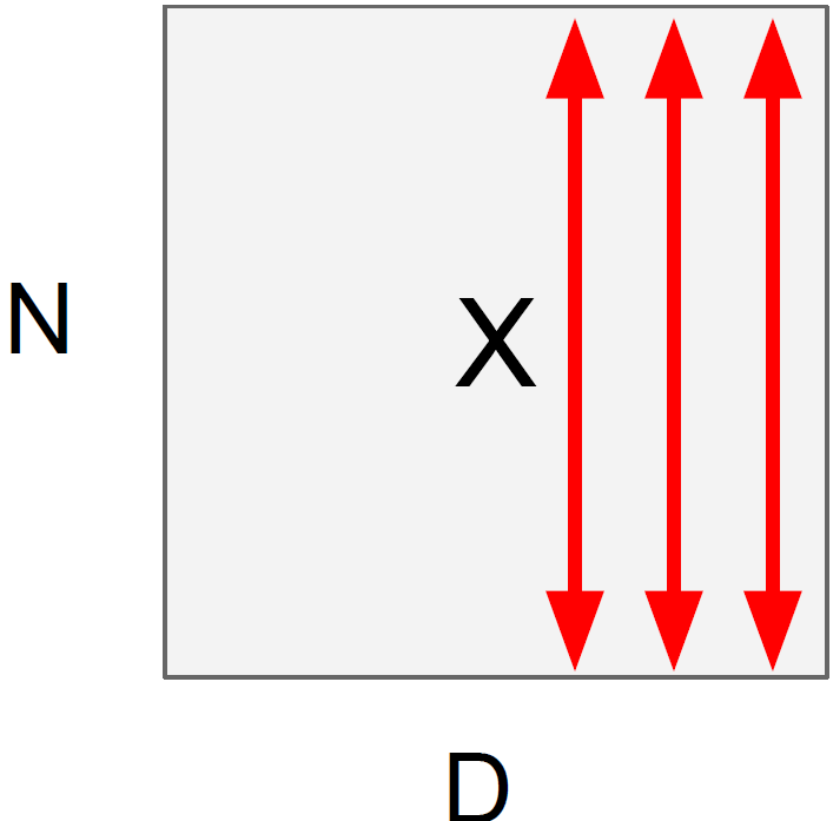
consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

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

this is a vanilla differentiable function...

Batch Normalization

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

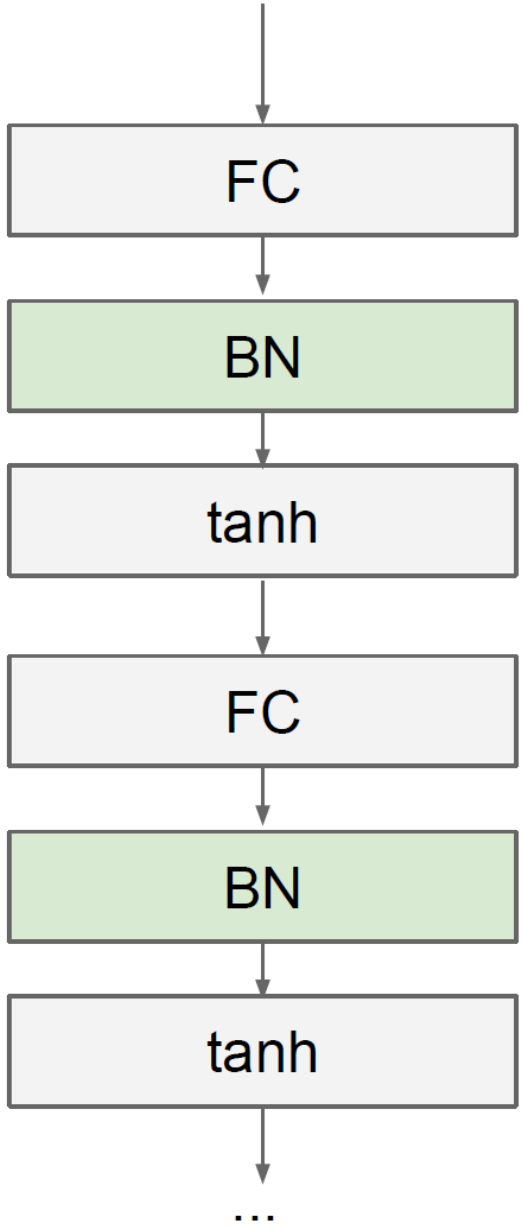


1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

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

Batch Normalization

Normalize:

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

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- 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

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

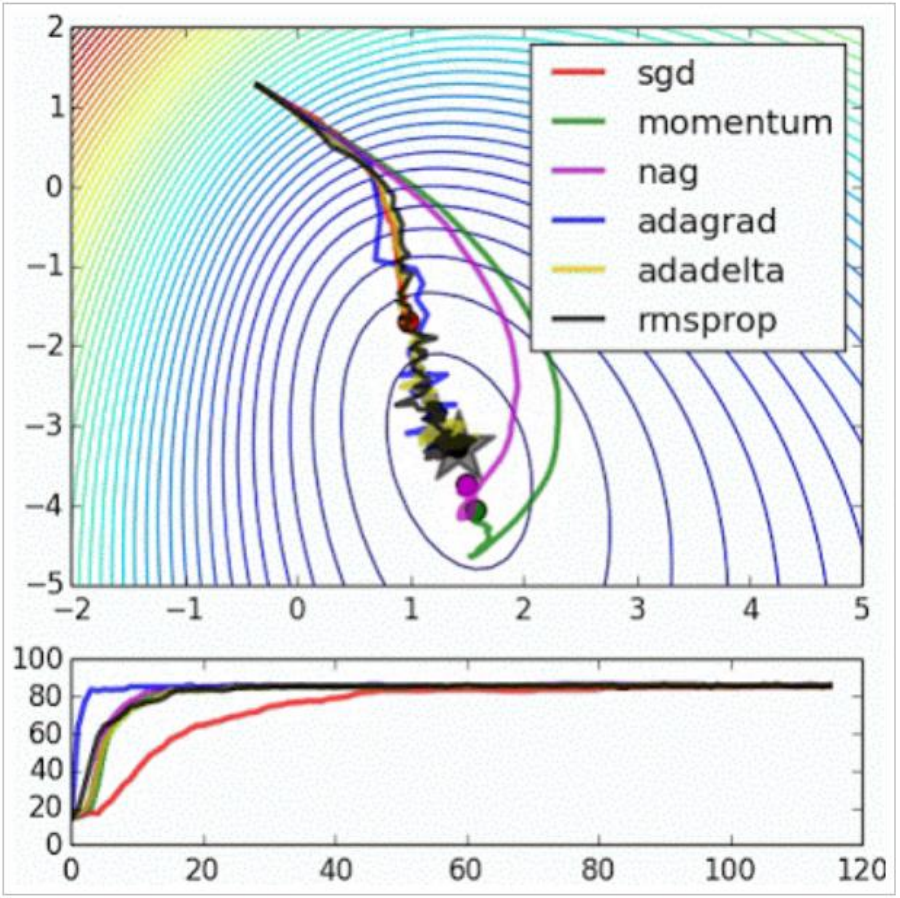
Note: at test time BatchNorm layer functions differently:

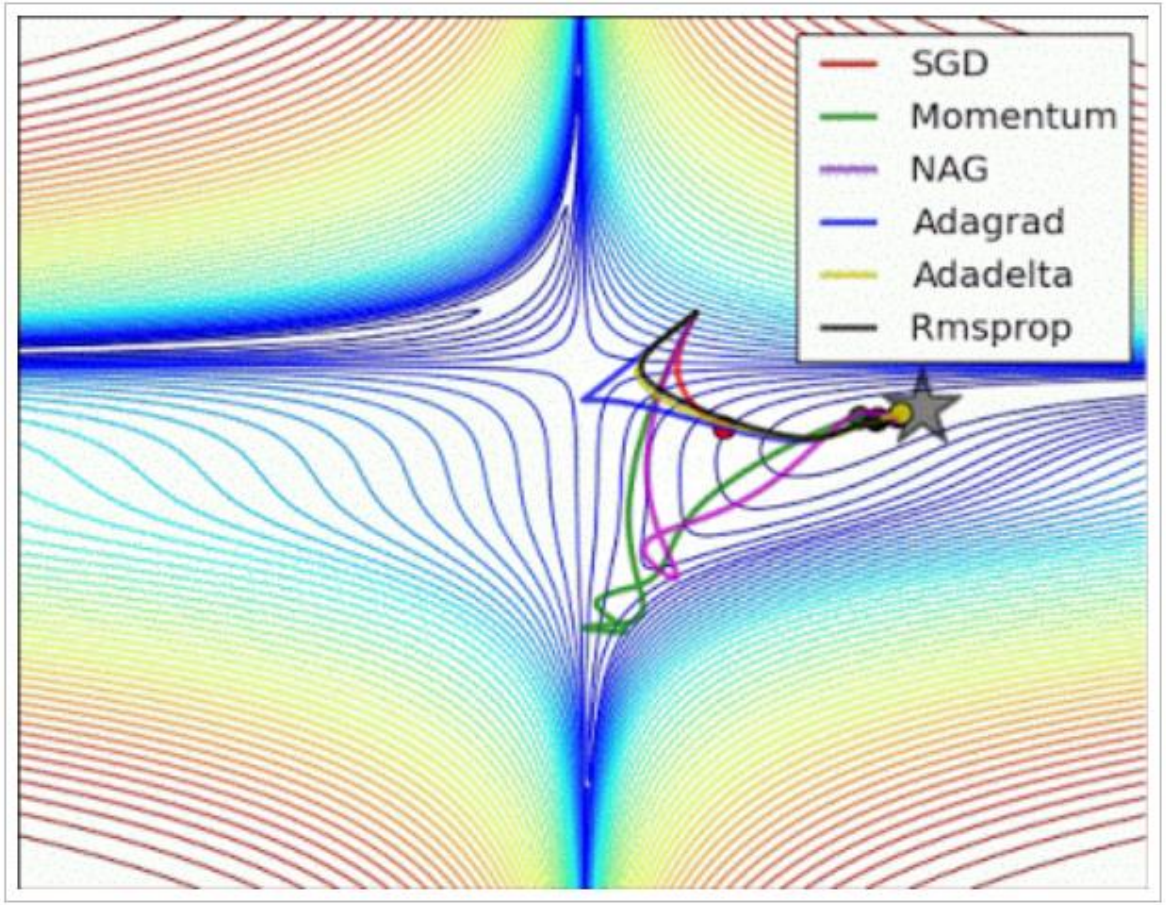
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Optimization Methods

Optimization Method

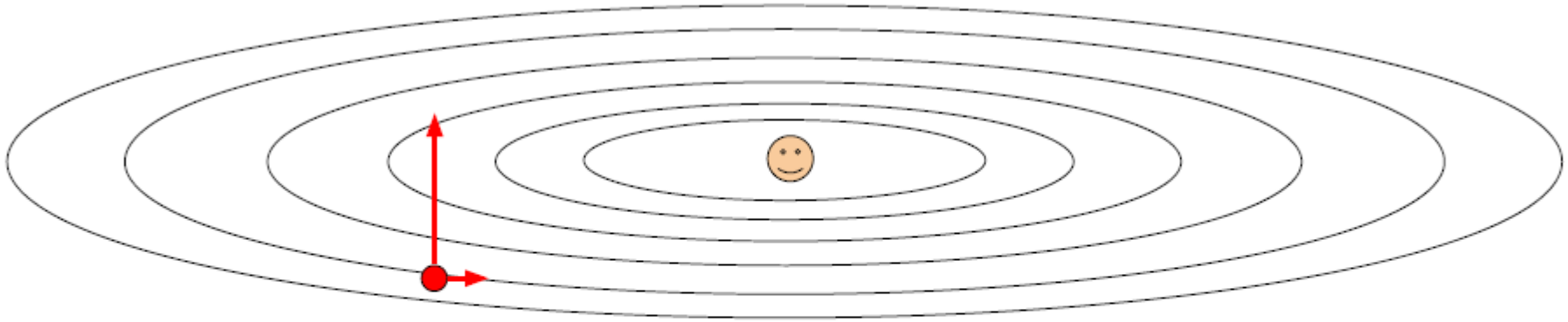




[Alec Radford's animations for optimization]

Gradient Descent Update

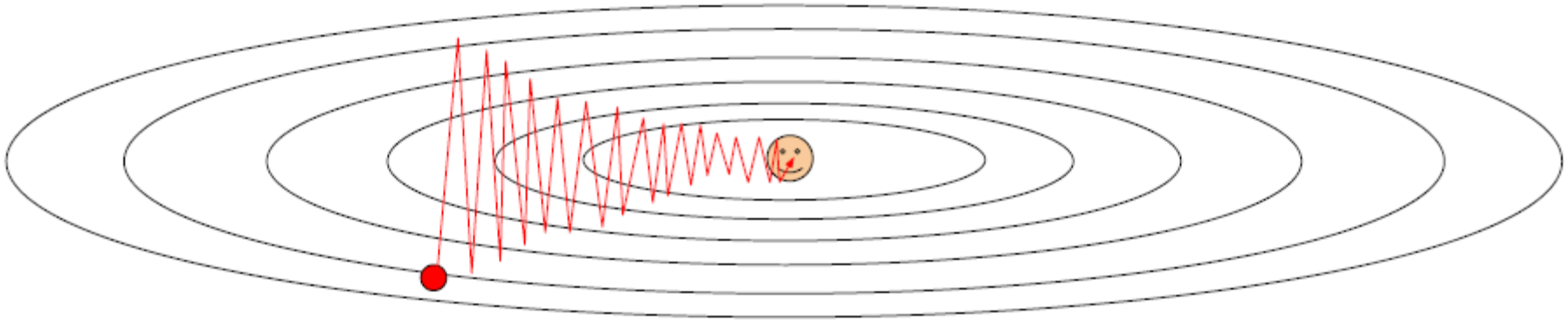
```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x -= learning_rate * dx
```



Q: What is the trajectory along which we converge towards the minimum with SGD?

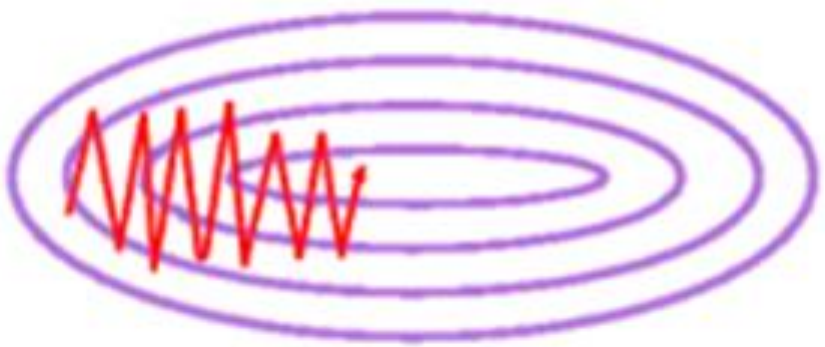
Gradient Descent Update

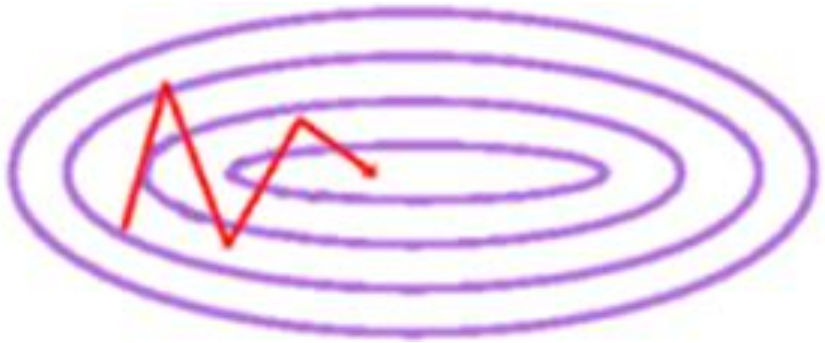
Suppose loss function is steep vertically but shallow horizontally:



Q: What is the trajectory along which we converge towards the minimum with SGD? very slow progress along flat direction, jitter along steep one

Momentum Update





```
# Vanilla gradient descent update
x -= learning_rate * dx
```

```
# Momentum update
v = mu * v - learning_rate * dx
x += v
```

Adagrad Update

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

Notice that the variable `cache` has size equal to the size of the gradient, and keeps track of per-parameter sum of squared gradients. This is then used to normalize the parameter update step, element-wise. Notice that the weights that receive high gradients will have their effective learning rate reduced, while weights that receive small or infrequent updates will have their effective learning rate increased. Amusingly, the square root operation turns out to be very important and without it the algorithm performs much worse. The smoothing term `eps` (usually set somewhere in range from 1e-4 to 1e-8) avoids division by zero. A downside of Adagrad is that in case of Deep Learning, the monotonic learning rate usually proves too aggressive and stops learning too early.

RMSProp Update

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

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

Here, `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999]. Notice that the `x+=` update is identical to Adagrad, but the `cache` variable is a "leaky". Hence, RMSProp still modulates the learning rate of each weight based on the magnitudes of its gradients, which has a beneficial equalizing effect, but unlike Adagrad the updates do not get monotonically smaller.

Adam Update

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

Notice that the update looks exactly as RMSProp update, except the "smooth" version of the gradient `m` is used instead of the raw (and perhaps noisy) gradient vector `dx`. Recommended values in the paper are `eps = 1e-8`, `beta1 = 0.9`, `beta2 = 0.999`. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative. The full Adam update also includes a *bias correction* mechanism, which compensates for the fact that in the first few time steps the vectors `m,v` are both initialized and therefore biased at zero, before they fully "warm up". We refer the reader to the paper for the details, or the course slides where this is expanded on.

Adam Update

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

Momentum

Notice that the update looks exactly as RMSProp update, except the "smooth" version of the gradient `m` is used instead of the raw (and perhaps noisy) gradient vector `dx`. Recommended values in the paper are `eps = 1e-8`, `beta1 = 0.9`, `beta2 = 0.999`. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative. The full Adam update also includes a *bias correction* mechanism, which compensates for the fact that in the first few time steps the vectors `m,v` are both initialized and therefore biased at zero, before they fully "warm up". We refer the reader to the paper for the details, or the course slides where this is expanded on.

Adam Update

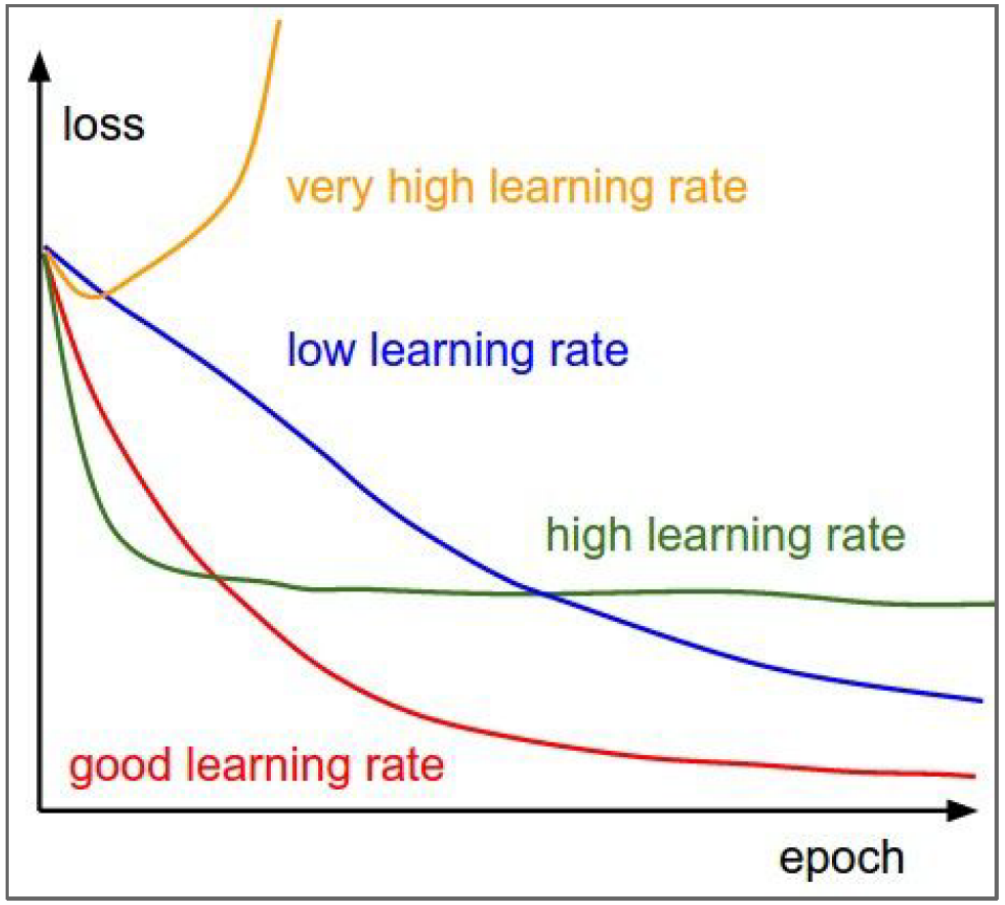
```
# Adam update
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x -= learning_rate * m (np.sqrt(v) + 1e-7)
```

Rmsprop like

Notice that the update looks exactly as RMSProp update, except the "smooth" version of the gradient `m` is used instead of the raw (and perhaps noisy) gradient vector `dx`. Recommended values in the paper are `eps = 1e-8`, `beta1 = 0.9`, `beta2 = 0.999`. In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative. The full Adam update also includes a *bias correction* mechanism, which compensates for the fact that in the first few time steps the vectors `m,v` are both initialized and therefore biased at zero, before they fully "warm up". We refer the reader to the paper for the details, or the course slides where this is expanded on.

Learning Rate

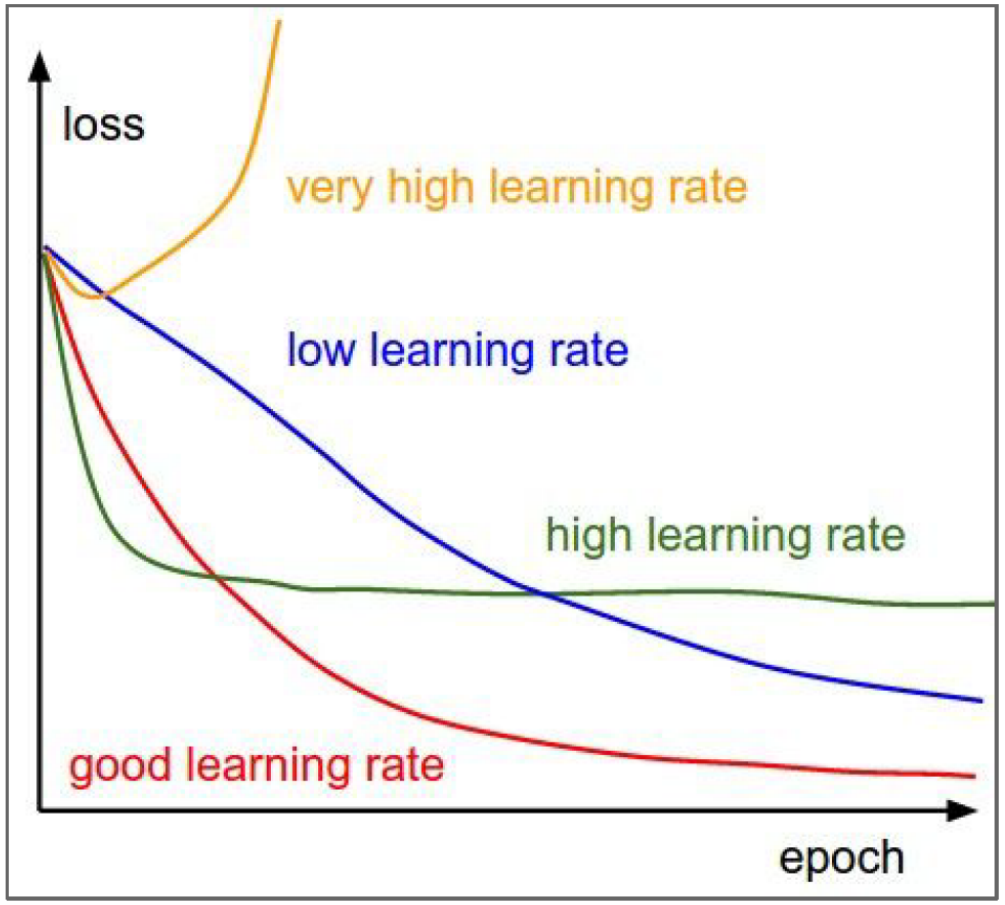
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning Rate

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

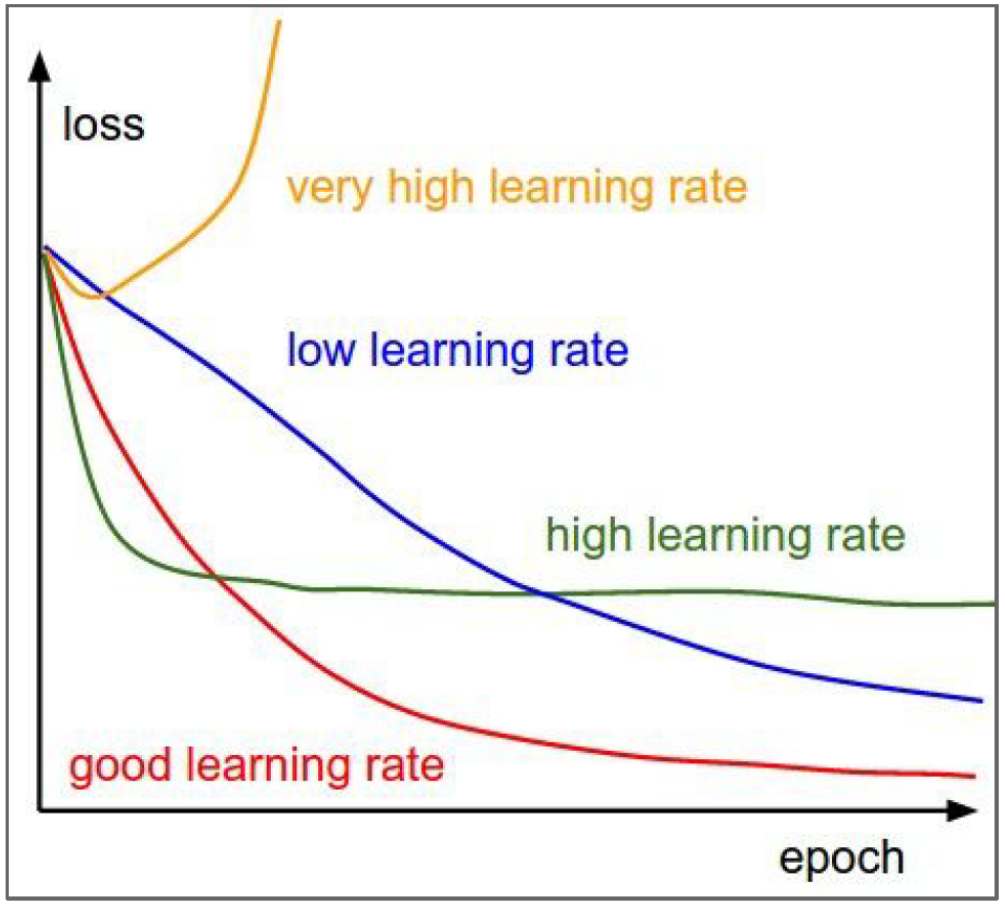
$$\alpha = \alpha_0 e^{-kt}$$

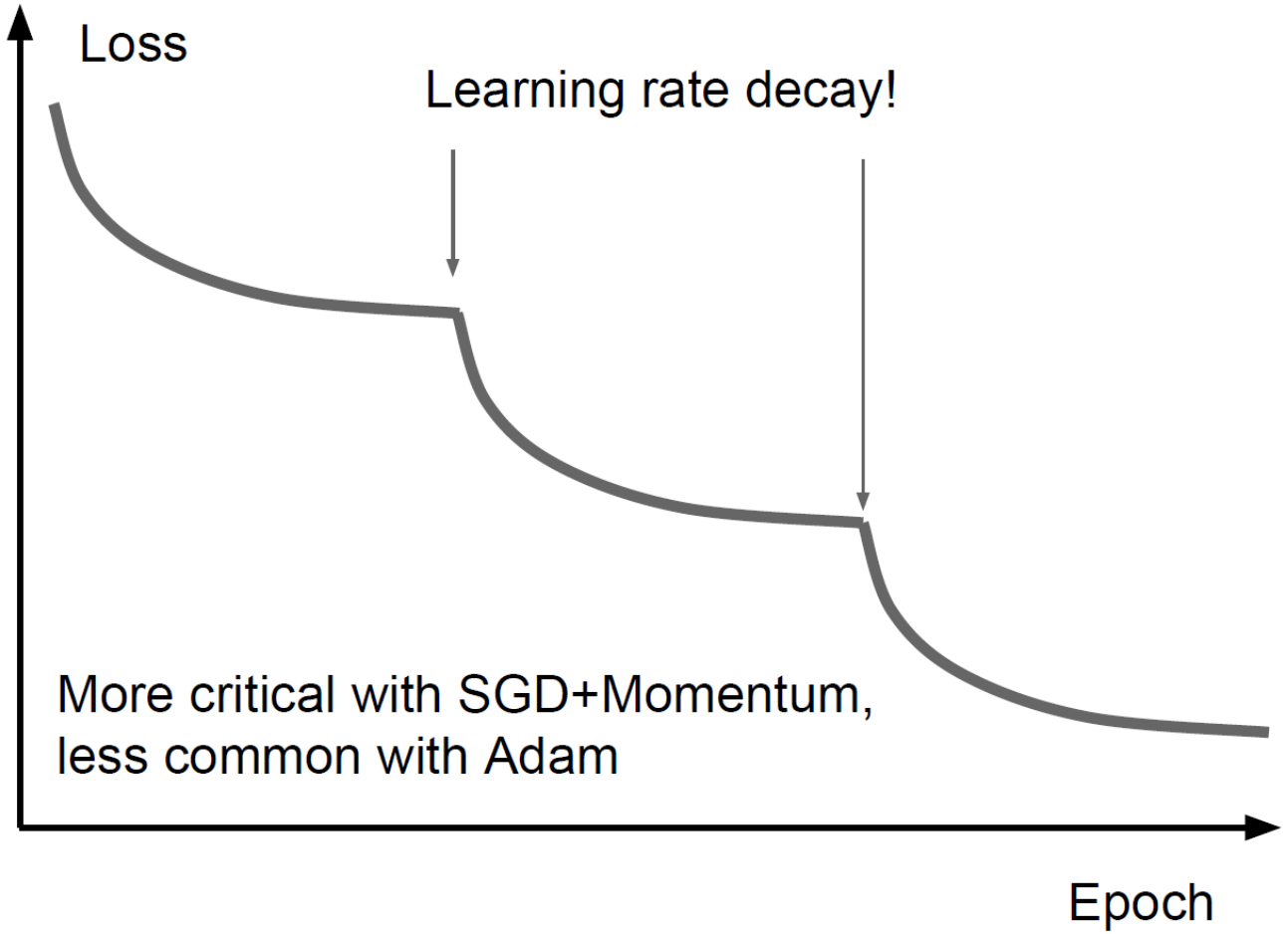
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

Learning Rate

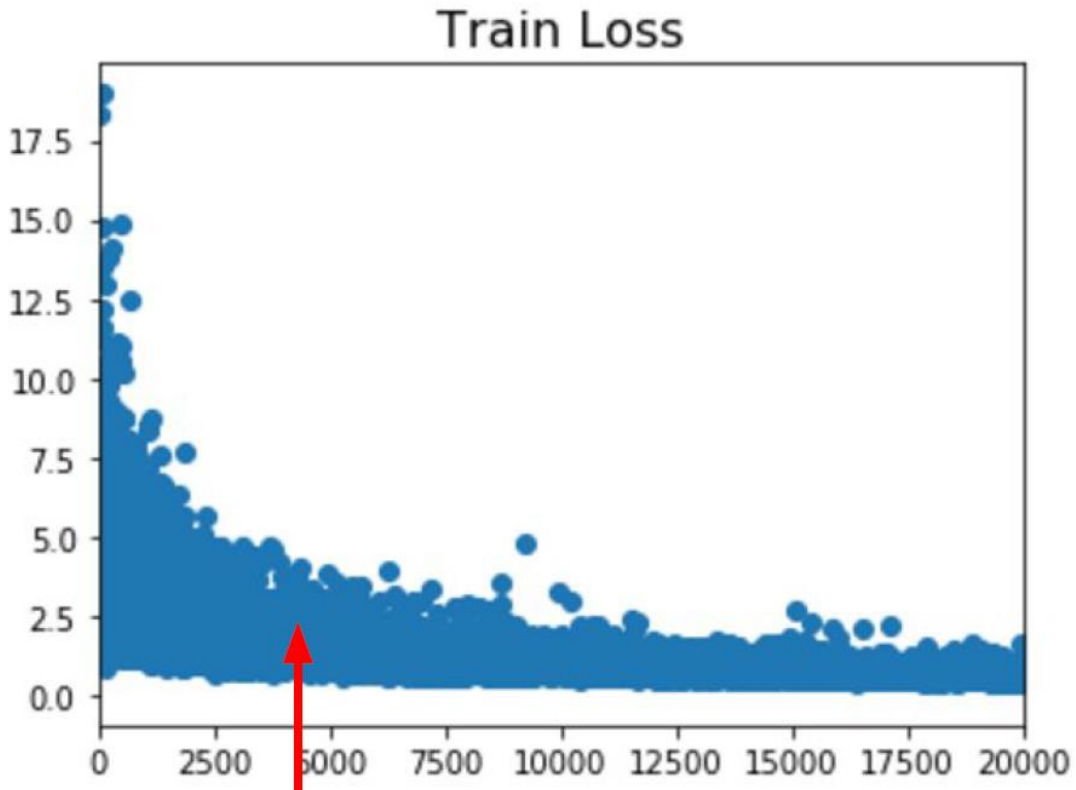
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

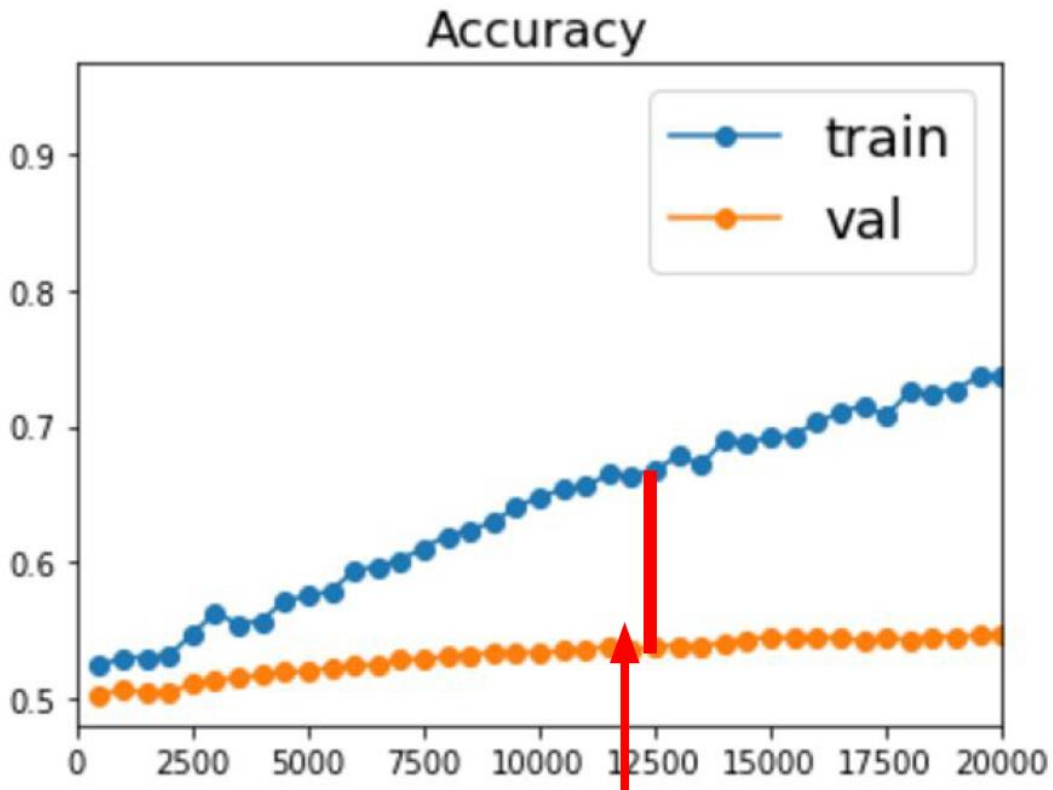




Ensemble and Regularization

Beyond Training Error





Better optimization algorithms help reduce training loss

But we really care about error on new data - how to reduce the gap?

Model Ensembles

1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Regularization: Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

In common use:

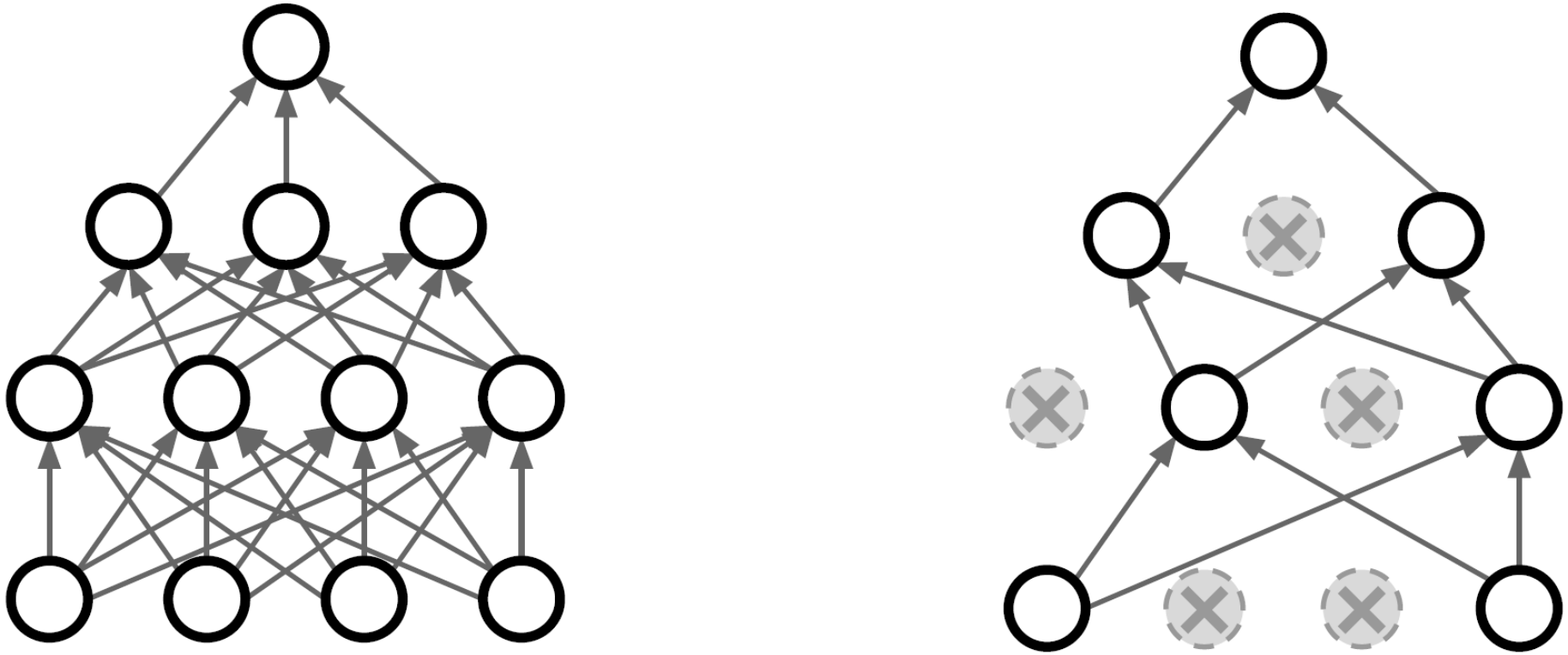
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization: Dropout

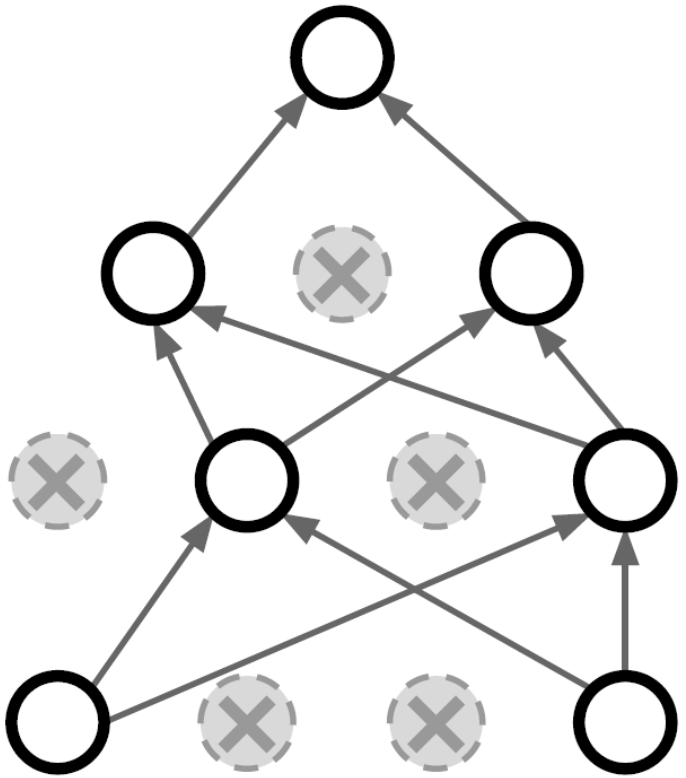
In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

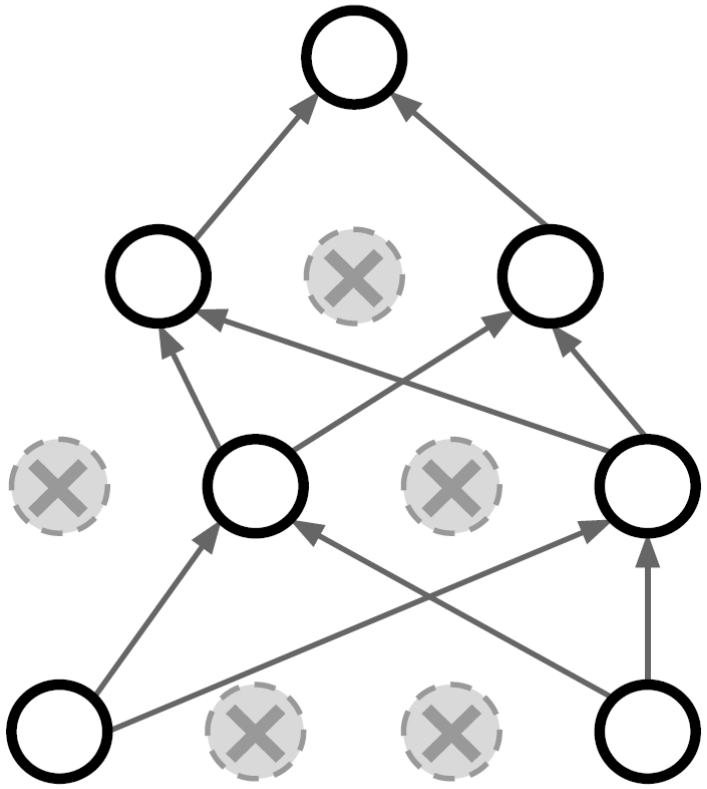
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

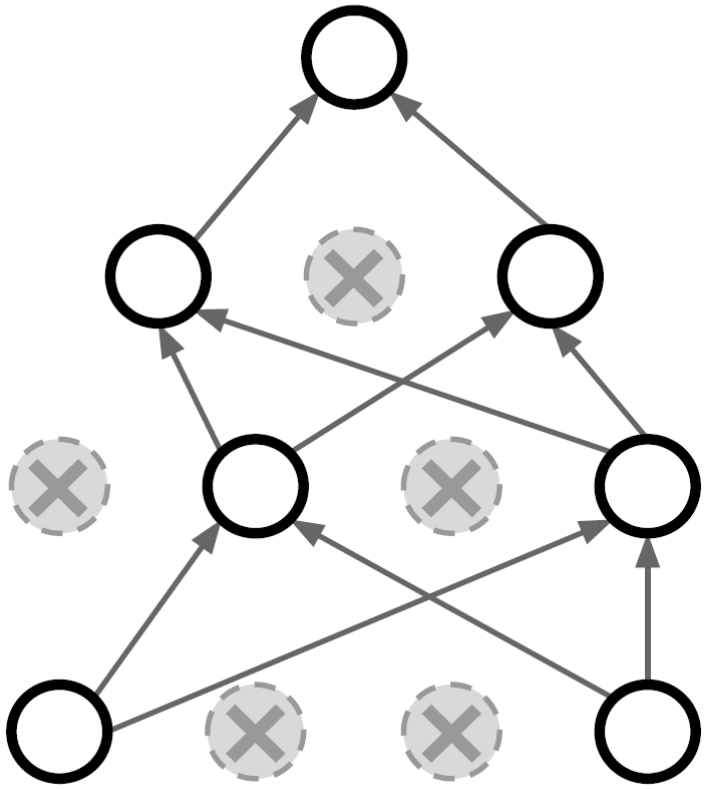
How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Regularization: Dropout

How can 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

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Regularization: Dropout

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

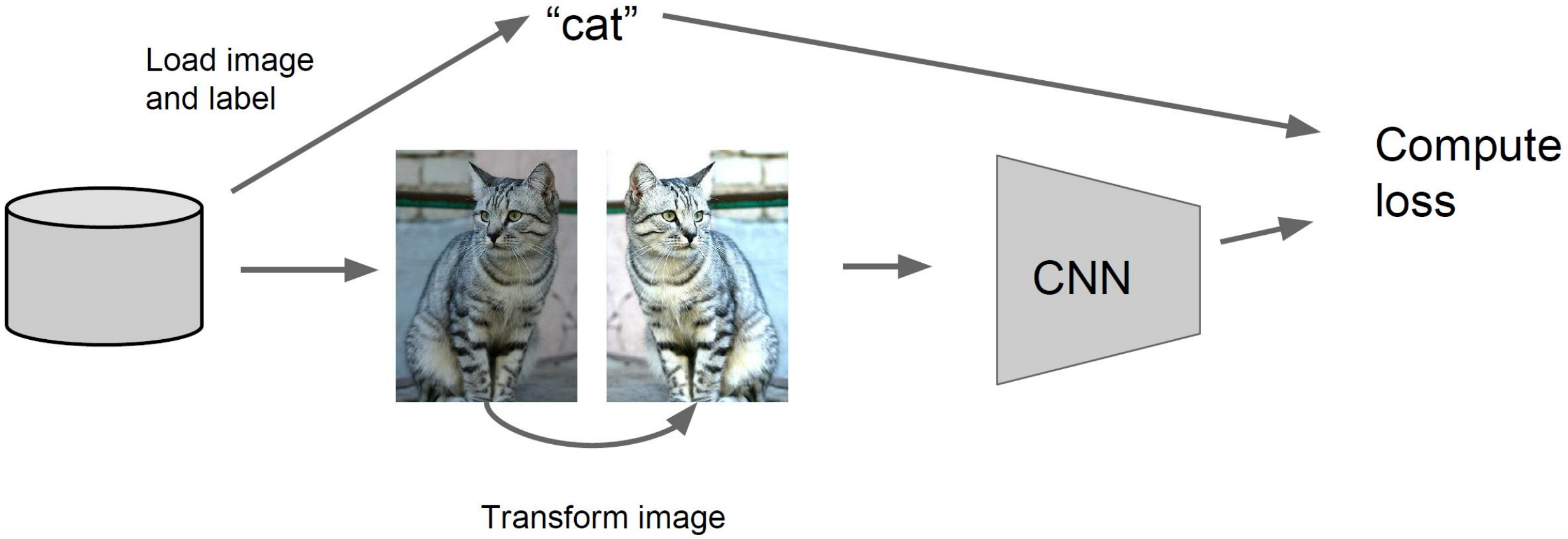
Data Augmentation

Regularization: Dropout

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

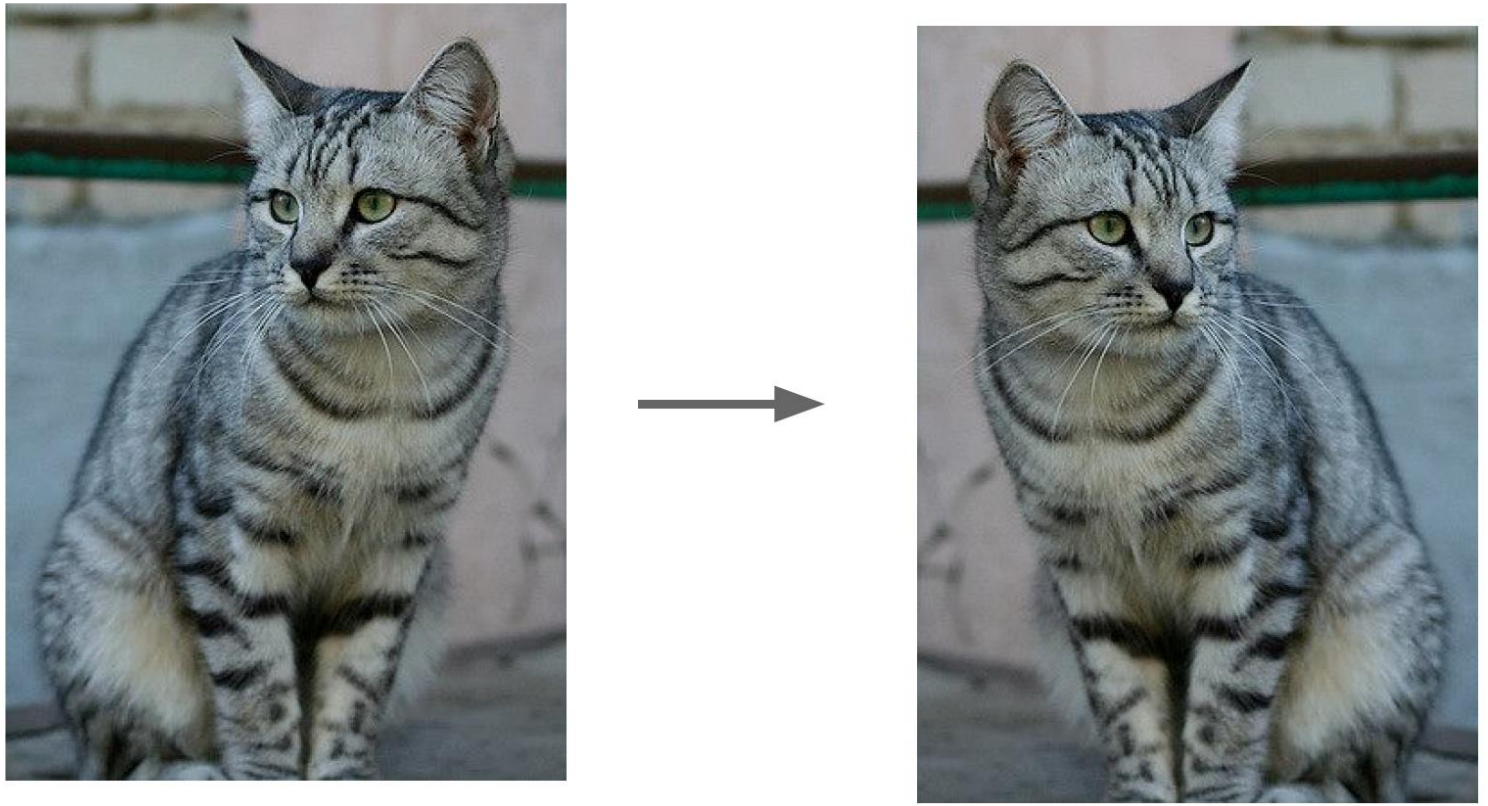
At test time all neurons are active always
=> We must scale the activations so that for each neuron:
output at test time = expected output at training time

Regularization: Data Augmentation



Regularization: Data Augmentation

Horizontal Flips



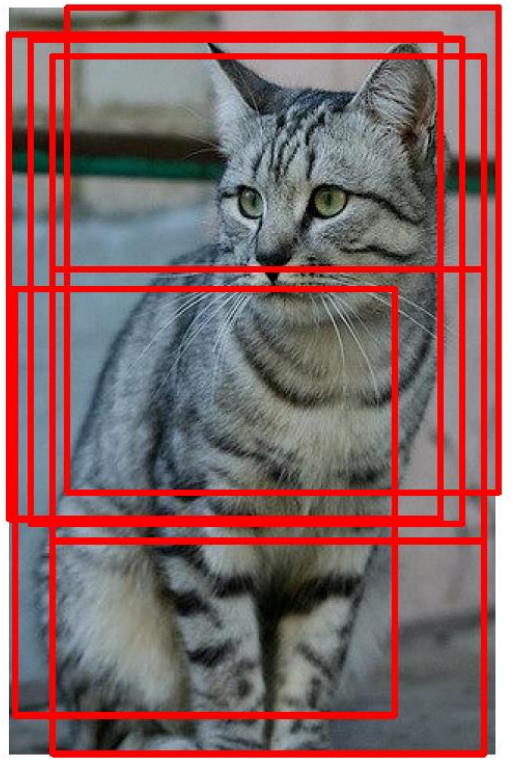
Regularization: Data Augmentation

Random crops and 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



Regularization: Data Augmentation

Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

References

Stanford University CS231n: Convolutional Neural Networks for Visual Recognition

Deep Learning Summer School, Montreal 2016 - VideoLectures.NET

Stanford University CS224d: Deep Learning for Natural Language Processing