Neural Neural Networks Part II

Suleyman Demirel University

CSS634: Deep Learning

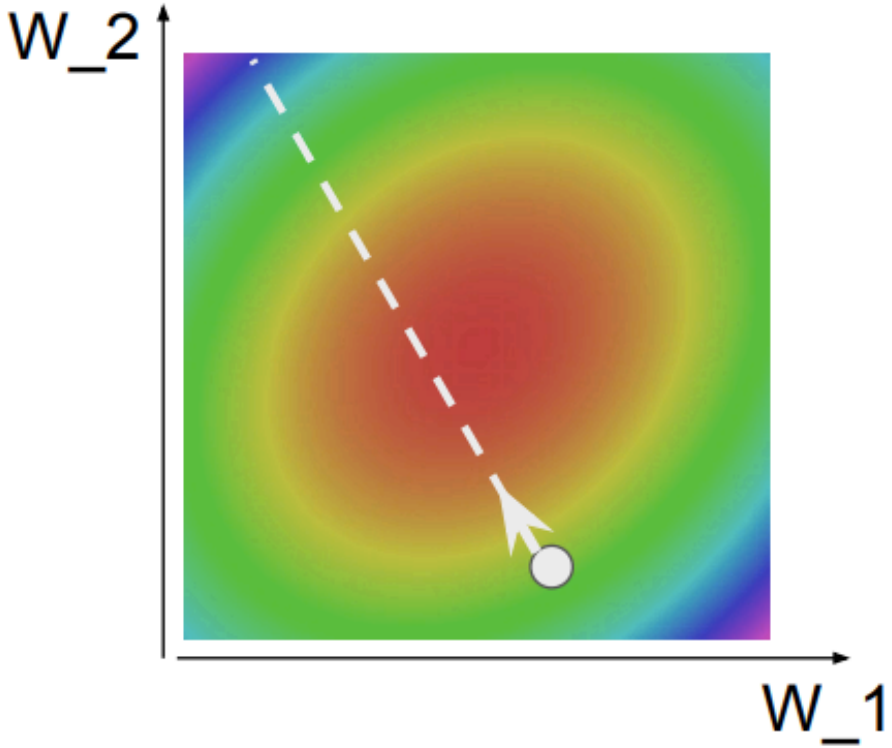
PhD Abay Nussipbekov

Part II

- Optimization
  - Problems with SGD
  - Exponentially weighted averages
  - Gradient descent with momentum
  - RMSprop
  - Adam optimization
  - Learning rate decay
- Regularization
  - More data
  - Early stopping
  - L1/L2 regularization
  - Dropout
  - Model ensembles
- Transfer Learning

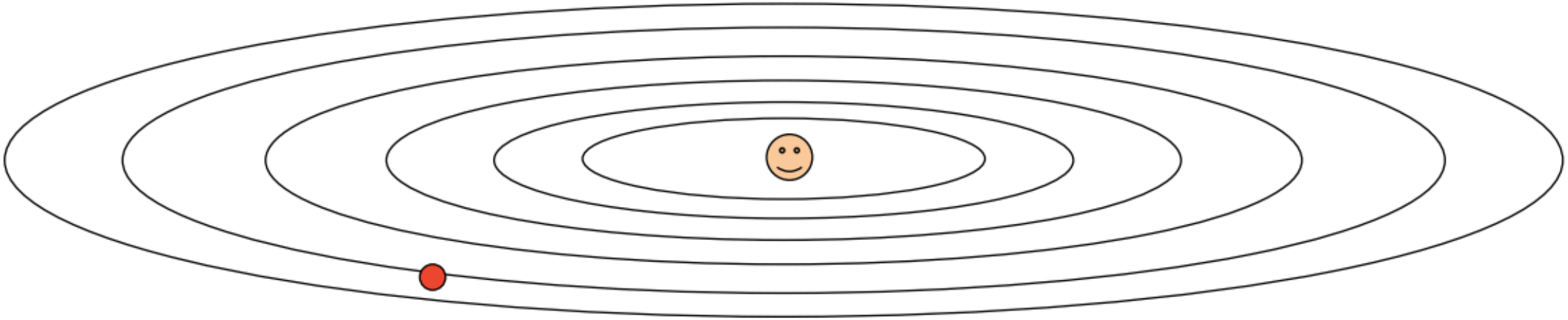
Optimization

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Optimization: Problems with SGD

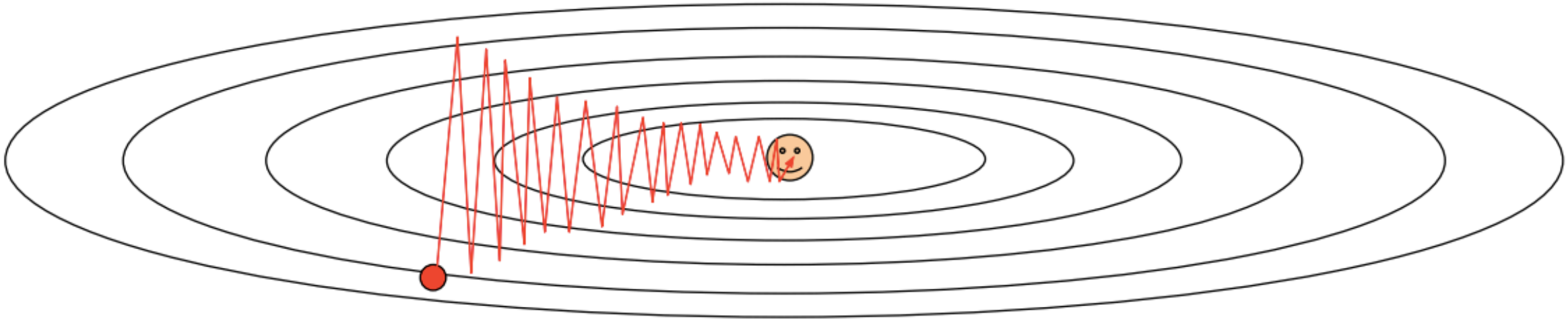
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



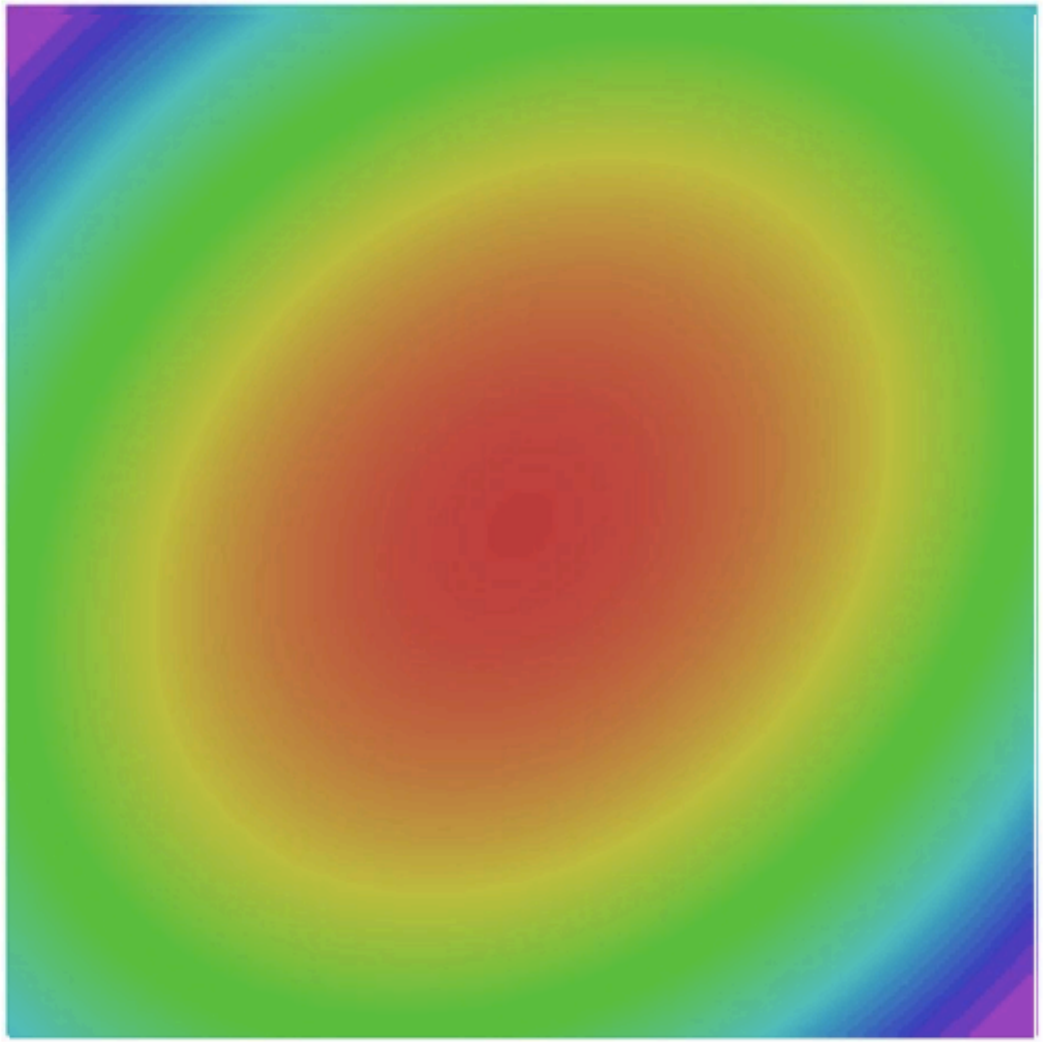
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

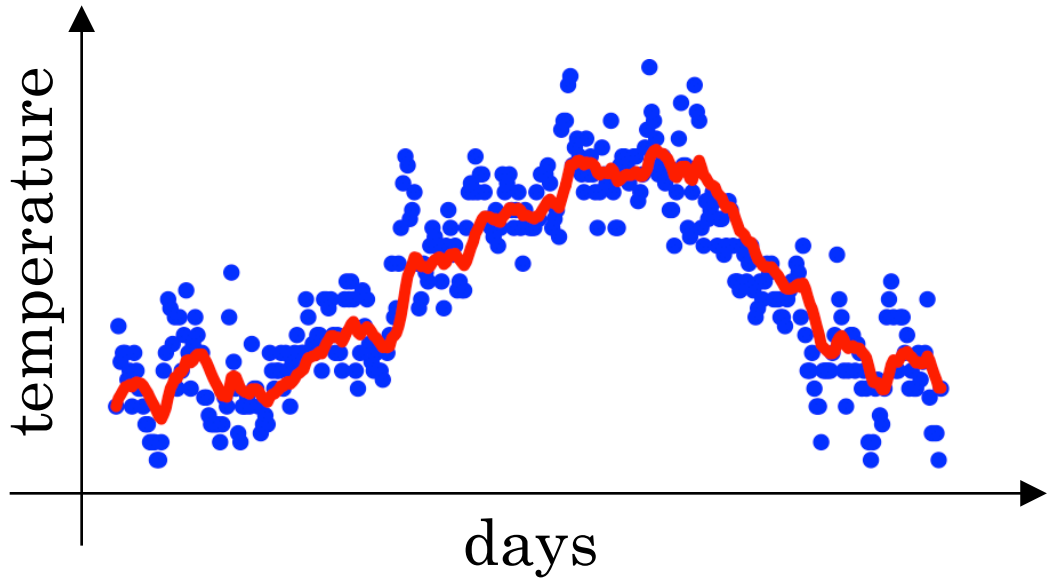
$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Exponentially Weighted Averages

Temperature Measure

$$\theta_1 = 40°\text{F}$$
$$\theta_2 = 49°\text{F}$$
$$\theta_3 = 45°\text{F}$$
$$\vdots$$
$$\theta_{180} = 60°\text{F}$$
$$\theta_{181} = 56°\text{F}$$
$$\vdots$$



$$v_0 = 0$$
$$v_1 = 0.9\, v_0 + 0.1\, \theta_1$$
$$v_2 = 0.9\, v_1 + 0.1\, \theta_2$$
$$v_3 = 0.9\, v_2 + 0.1\, \theta_3$$
$$\vdots$$
$$v_t = 0.9\, v_{t-1} + 0.1\, \theta_t$$

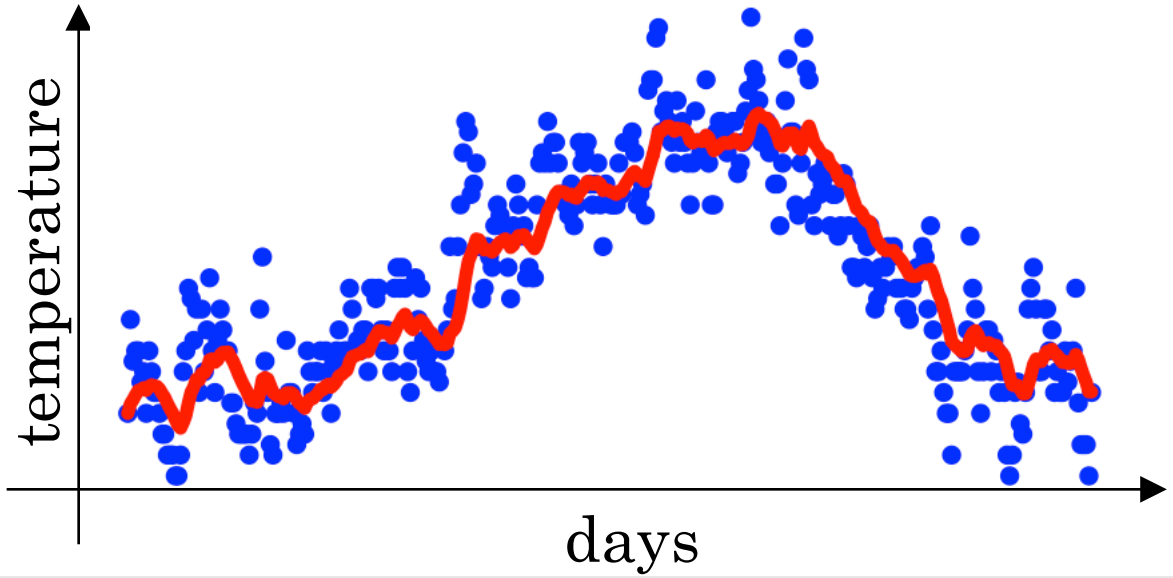
Exponentially Weighted Averages

$$v_t = \beta\, v_{t-1} + (1-\beta)\, \theta_t$$

$\beta = 0.9 : \approx 10\ days\ temperature$

v_t approximately averaging over $\approx \frac{1}{1-\beta}$

days temperature



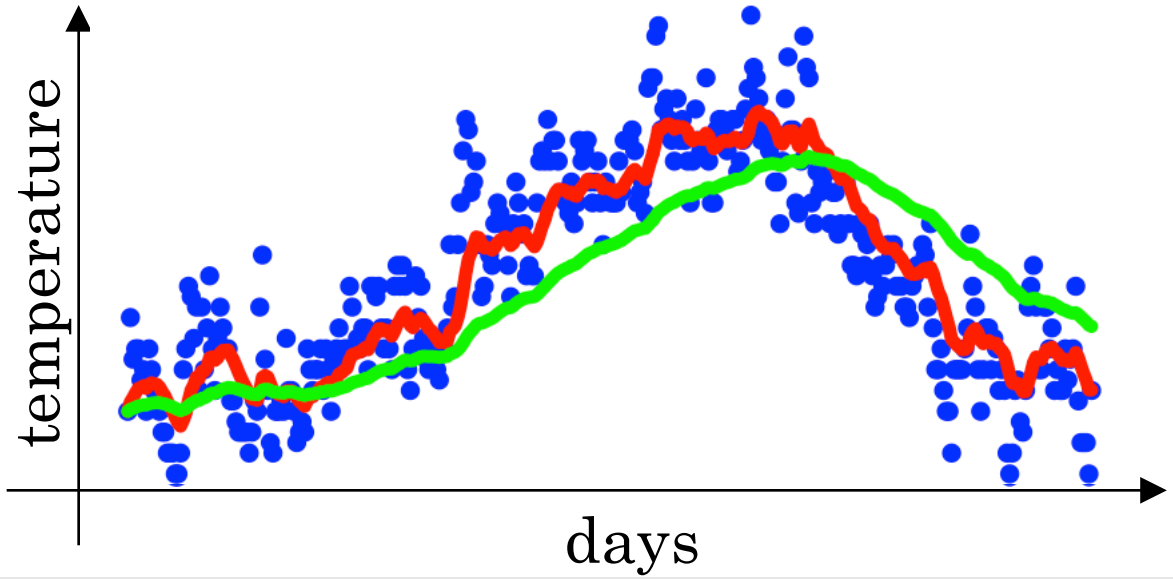
Exponentially Weighted Averages

$$v_t = \beta\, v_{t-1} + (1-\beta)\, \theta_t$$

$\beta = 0.9 : \approx 10$ *days temperature*

$\beta = 0.98 : \approx 50$ *days temperature*

v_t approximately averaging over $\approx \frac{1}{1-\beta}$ days temperature



Exponentially Weighted Averages

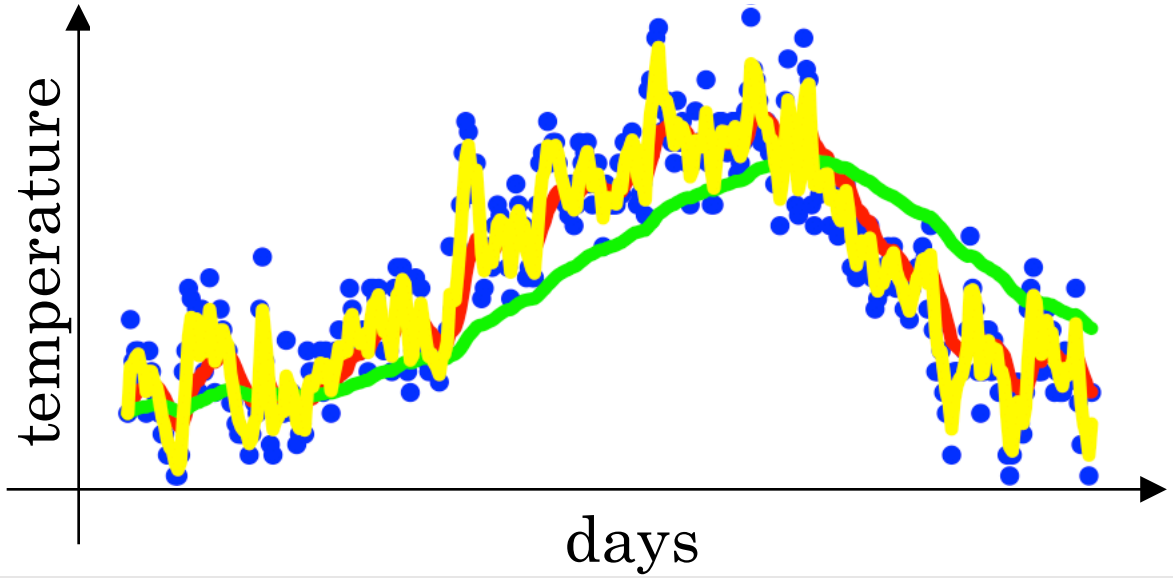
$$v_t = \beta\, v_{t-1} + (1-\beta)\,\theta_t$$

$\beta = 0.9 : \approx 10\ days\ temperature$

$\beta = 0.98 : \approx 50\ days\ temperature$

$\beta = 0.5 : \approx 2\ days\ temperature$

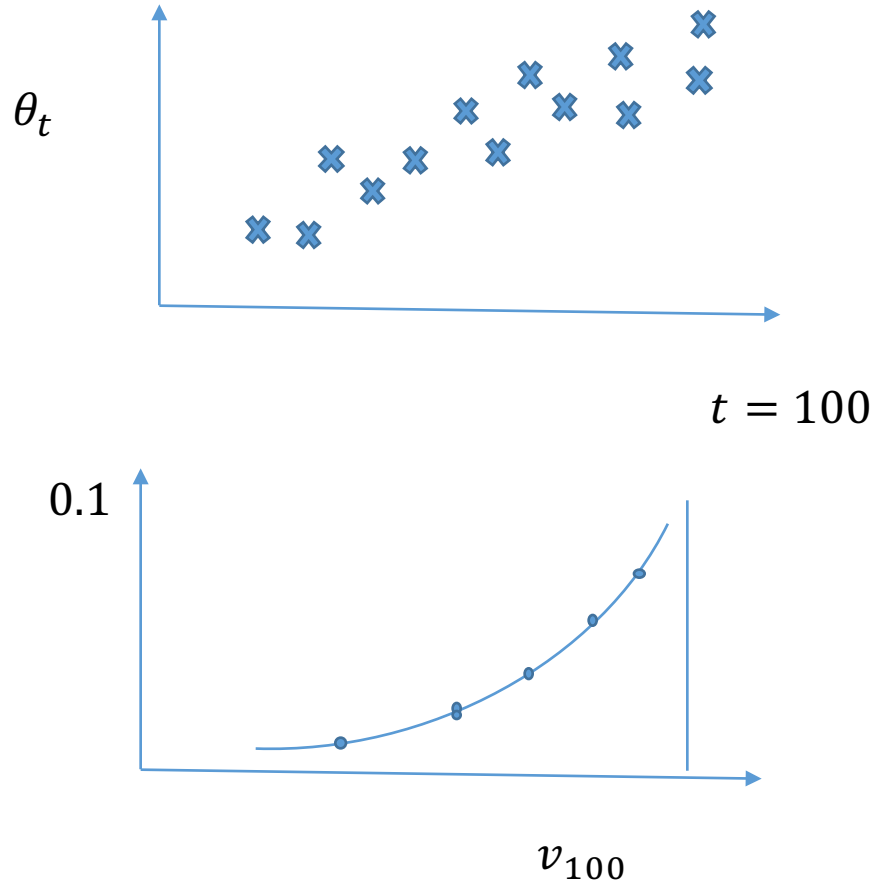
v_t approximately averaging over $\approx \frac{1}{1-\beta}$ days temperature



Understanding Exponentially Weighted Averages

$$v_t = \beta\, v_{t-1} + (1-\beta)\, \theta_t$$

$$v_{100} = 0.9\, v_{99} + 0.1\, \theta_{100}$$
$$v_{99} = 0.9\, v_{98} + 0.1\, \theta_{99}$$
$$v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98}$$
$$\vdots$$



$$v_{100} = 0.1\, \theta_{100} + 0.9\, (0.1\, \theta_{99} + 0.9\, (0.1\, \theta_{98} + 0.9\, (\ldots)))$$
$$= 0.1\, \theta_{100} + 0.1 * 0.9\, \theta_{99} + 0.1 * (0.9)^2\, \theta_{98} + 0.1 * (0.9)^3\, \theta_{97} + \ldots$$

General Algorithm

$v_0 = 0$

Repeat {

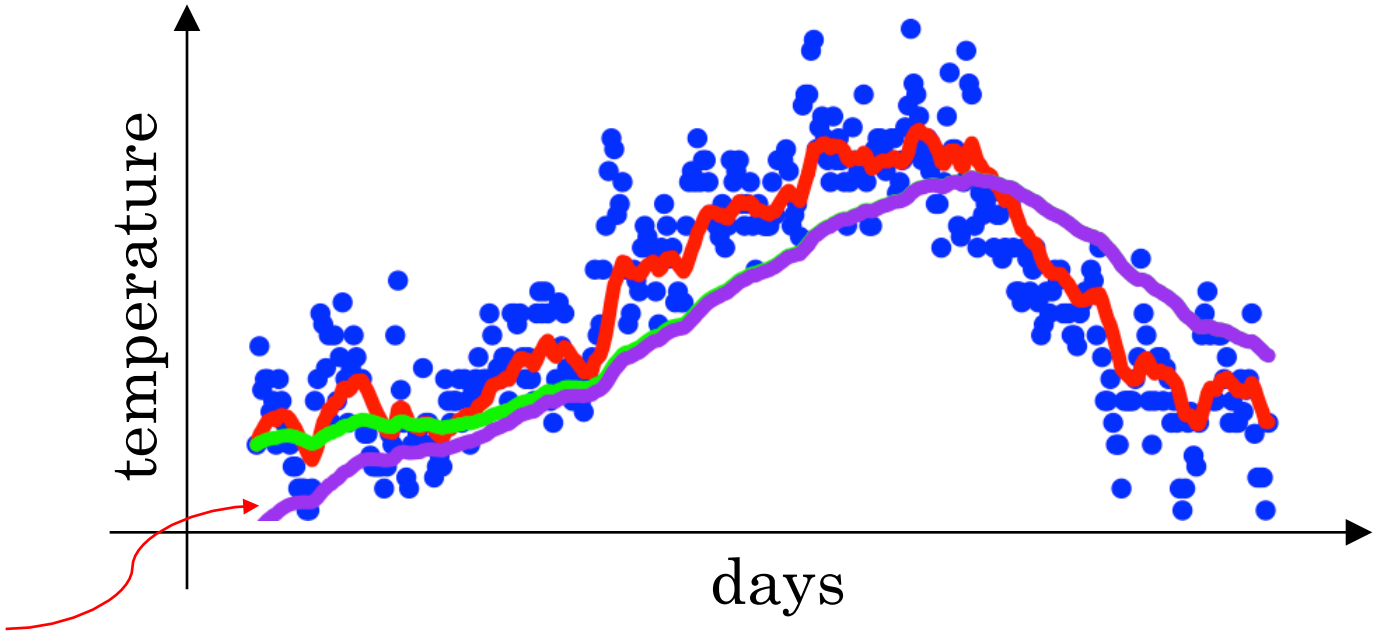
Get next θ_t

$v_\theta := \beta\ v_\theta + (1 - \beta)\theta_t$

}

This method is computationally and memory more efficient compared to always calculating the average of last 10 (for example) elements.

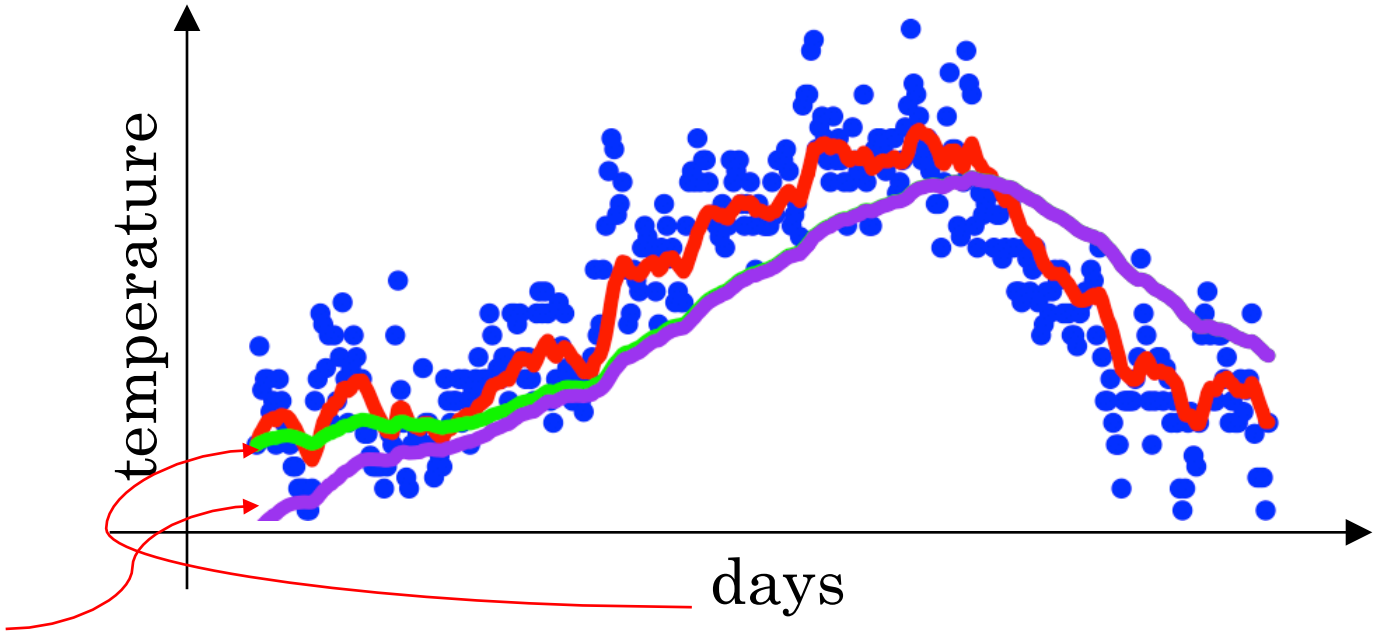
Bias Correction in Exponentially Weighted Averages



$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$v_0 = 0$$
$$v_1 = 0.98\, v_0 + 0.02\, \theta_1$$
$$v_2 = 0.98\, v_1 + 0.02\, \theta_2$$
$$= 0.98 * 0.02\, \theta_1 + 0.02\, \theta_2$$
$$= 0.0196\, \theta_1 + 0.02\, \theta_2$$

Bias Correction in Exponentially Weighted Averages



$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$v_0 = 0$$
$$v_1 = 0.98\, v_0 + 0.02\, \theta_1$$
$$v_2 = 0.98\, v_1 + 0.02\, \theta_2$$
$$= 0.98 * 0.02\, \theta_1 + 0.02\, \theta_2$$
$$= 0.0196\, \theta_1 + 0.02\, \theta_2$$

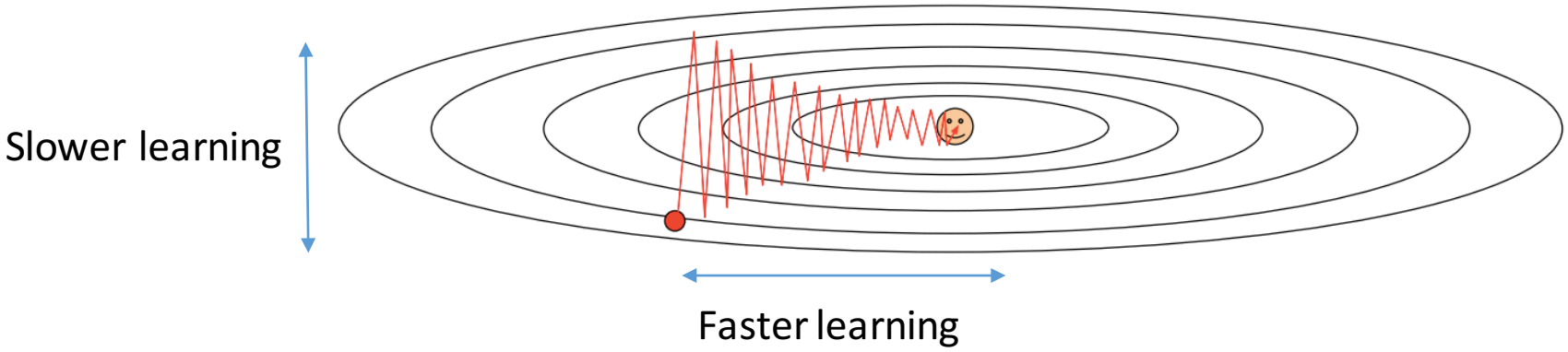
$$\frac{v_t}{1-\beta^t}$$

$$t = 2: \quad 1-\beta^t = 1-0.98^2 = 0.0396$$

$$\frac{v_2}{0.0396} = \frac{0.0196\, \theta_1 + 0.02\, \theta_2}{0.0396}$$

SGD + Momentum

Gradient Descent with Momentum



Momentum:

On iteration t:

Compute $\nabla w_{i,j}$ on current mini-batch

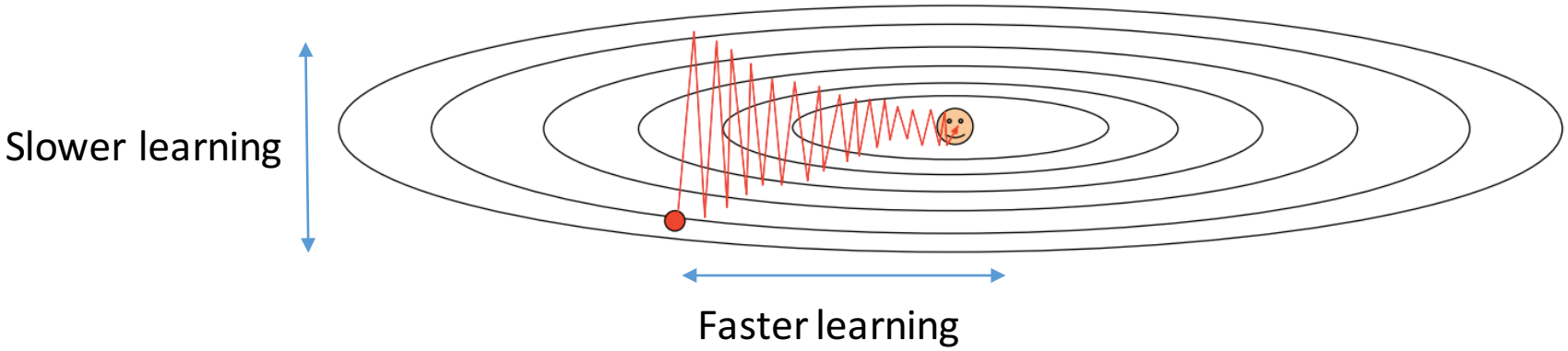
$$v_{\nabla w_{i,j}} = \beta\, v_{\nabla w_{i,j}} + (1-\beta)\nabla w_{i,j}$$

$$w_{i,j} := w_{i,j} - \alpha\, v_{\nabla w_{i,j}}$$

→ exponentially weighted average of gradient vector

$\beta = 0.9$ is a good default value (last 10 steps)

Gradient Descent with Momentum



Momentum:

On iteration t:

Compute $\nabla w_{i,j}$ on current mini-batch

$$v_{\nabla w_{i,j}} = \beta\, v_{\nabla w_{i,j}} + (1-\beta)\nabla w_{i,j}$$

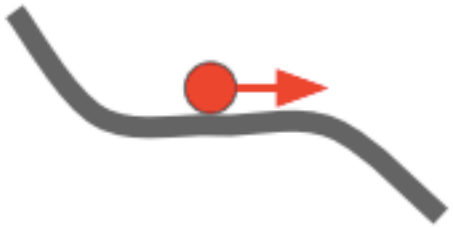
$$w_{i,j} := w_{i,j} - \alpha\, v_{\nabla w_{i,j}}$$

friction velocity acceleration

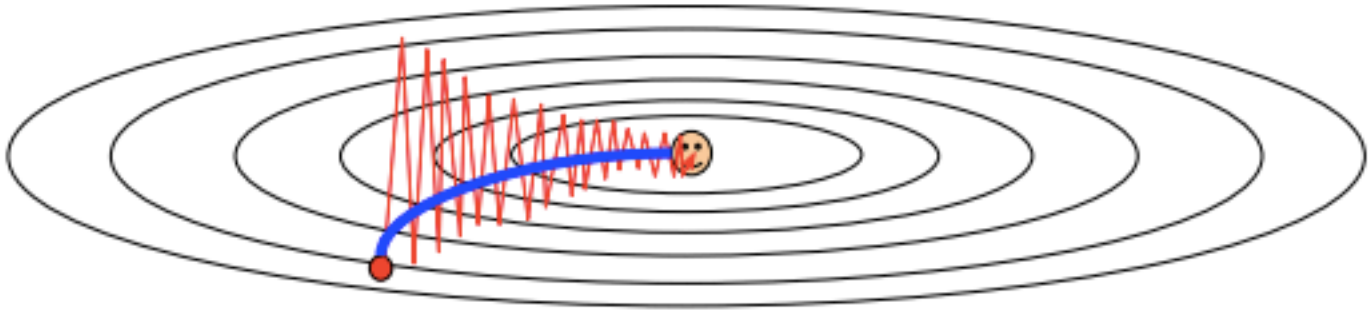
SGD + Momentum

Local Minima

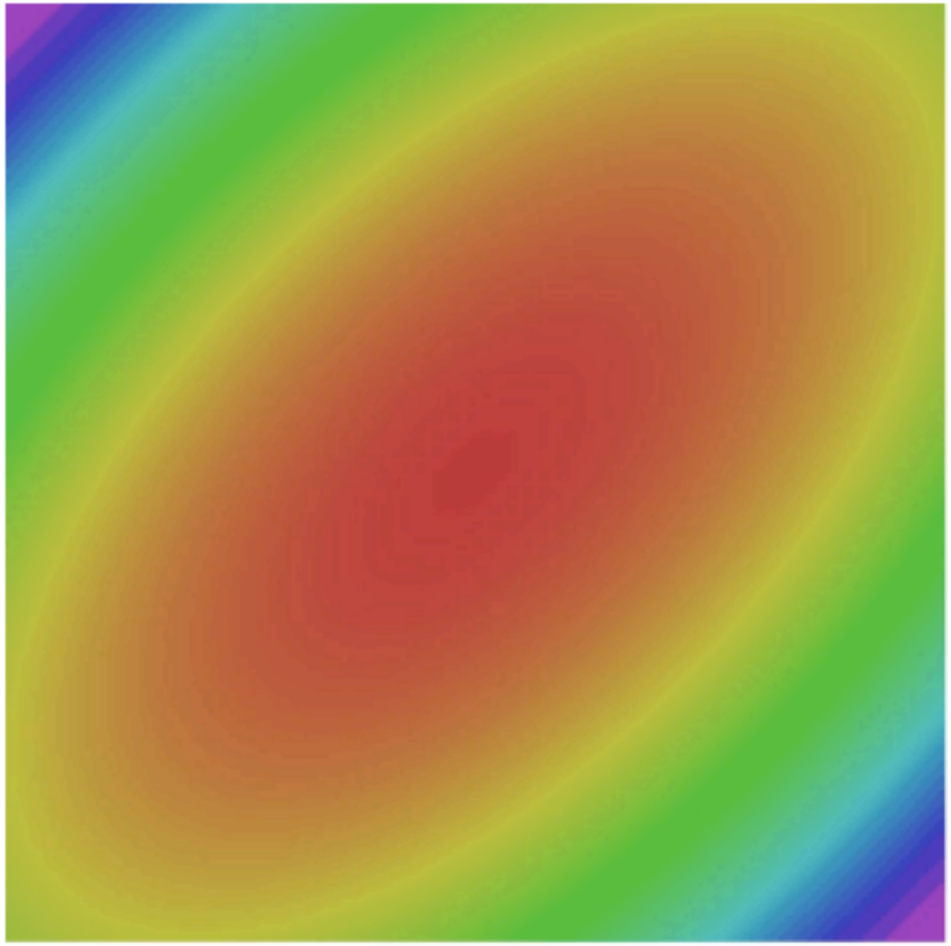
Saddle points



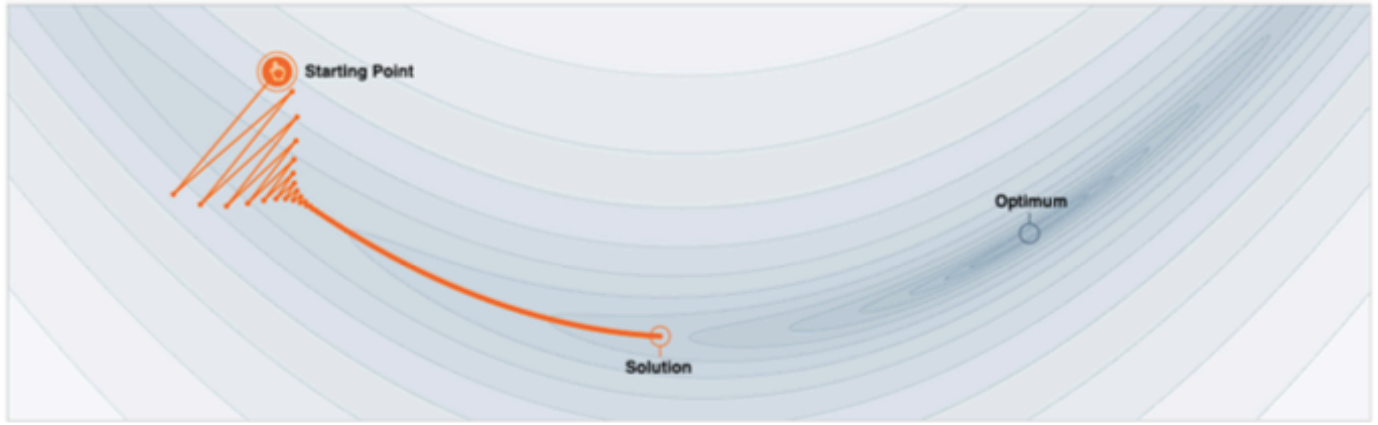
Poor Conditioning

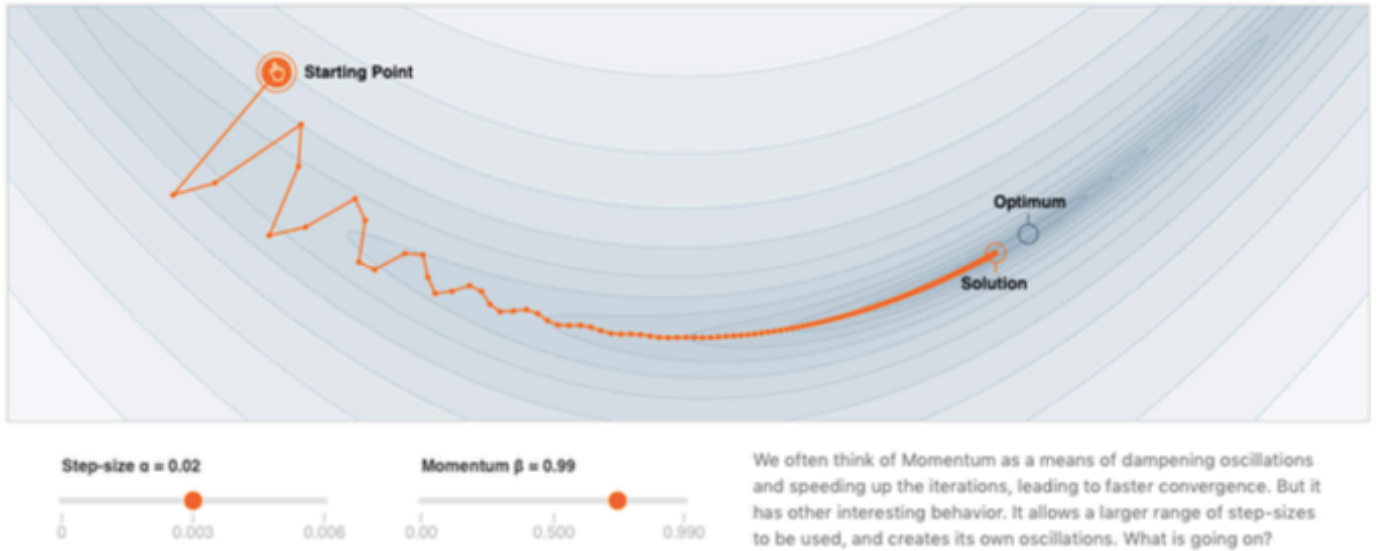


Gradient Noise



SGD SGD+Momentum





Source: https://distill.pub/2017/momentum/

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters:

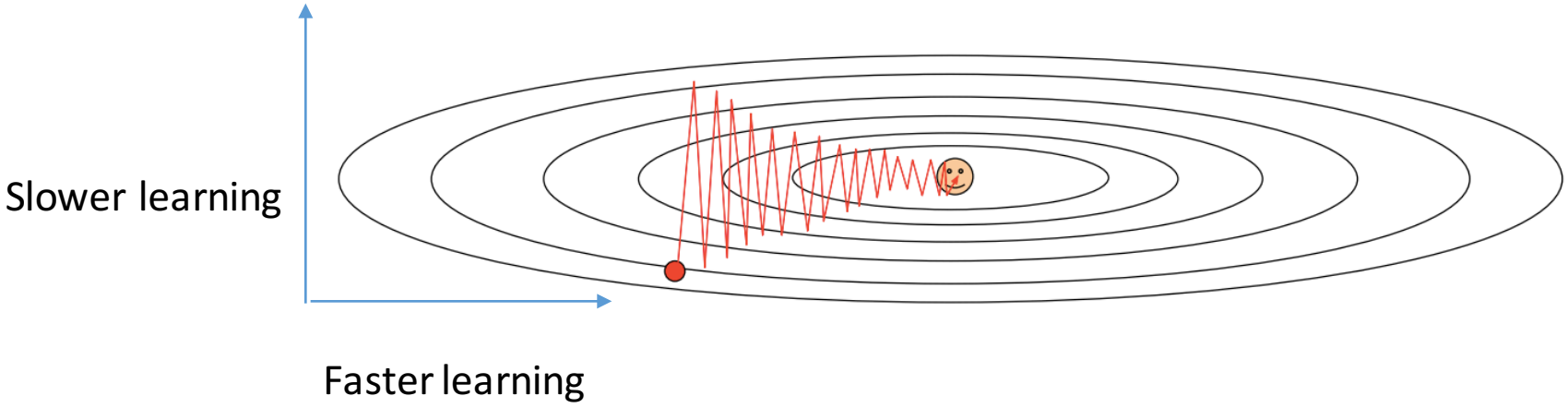
- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

Source: https://pytorch.org/docs/stable/optim.html

RMSprop

RMSprop



On iteration t:

Compute $\nabla w_{i,j}$ on current mini-batch

$$s_{\nabla w_{i,j}} = \beta_2 \, s_{\nabla w_{i,j}} + (1 - \beta_2)(\nabla w_{i,j})^2$$

Exponentially weighted average of the squares of the derivatives

$$w_{i,j} := w_{i,j} - \alpha \frac{\nabla w_{i,j}}{\sqrt{s_{\nabla w_{i,j}} + \varepsilon}}$$

The higher is the gradient the higher is it's denominator so it slows down and vice versa

Adam Optimization Algorithm

Adam Optimization Algorithm

$v_{\nabla w_{i,j}} = 0,\ s_{\nabla w_{i,j}} = 0$

On iteration t:

Compute $\nabla w_{i,j}$ on current mini-batch

$$v_{\nabla w_{i,j}} = \beta_1\ v_{\nabla w_{i,j}} + (1-\beta_1)\nabla w_{i,j}$$

momentum

$$s_{\nabla w_{i,j}} = \beta_2\ s_{\nabla w_{i,j}} + (1-\beta_2)(\nabla w_{i,j})^2$$

RMSprop

$$v_{\nabla w_{i,j}}^{corrected} = \frac{v_{\nabla w_{i,j}}}{(1-\beta_1^t)}$$

$$s_{\nabla w_{i,j}}^{corrected} = \frac{s_{\nabla w_{i,j}}}{(1-\beta_2^t)}$$

Bias correction

$$w_{i,j} := w_{i,j} - \alpha\ \frac{v_{\nabla w_{i,j}}^{corrected}}{\sqrt{s_{\nabla w_{i,j}}^{corrected}} + \varepsilon}$$

The higher is the gradient the higher is it's denominator so it slows down and vice versa

Adam Optimization Algorithm

- Commonly used learning algorithm that is proven to be very effective for many different NN of a wide variety of architectures
- Hyperparamter choice:

  α: needs to be tune

  β_1: 0.9

  β_2: 0.999

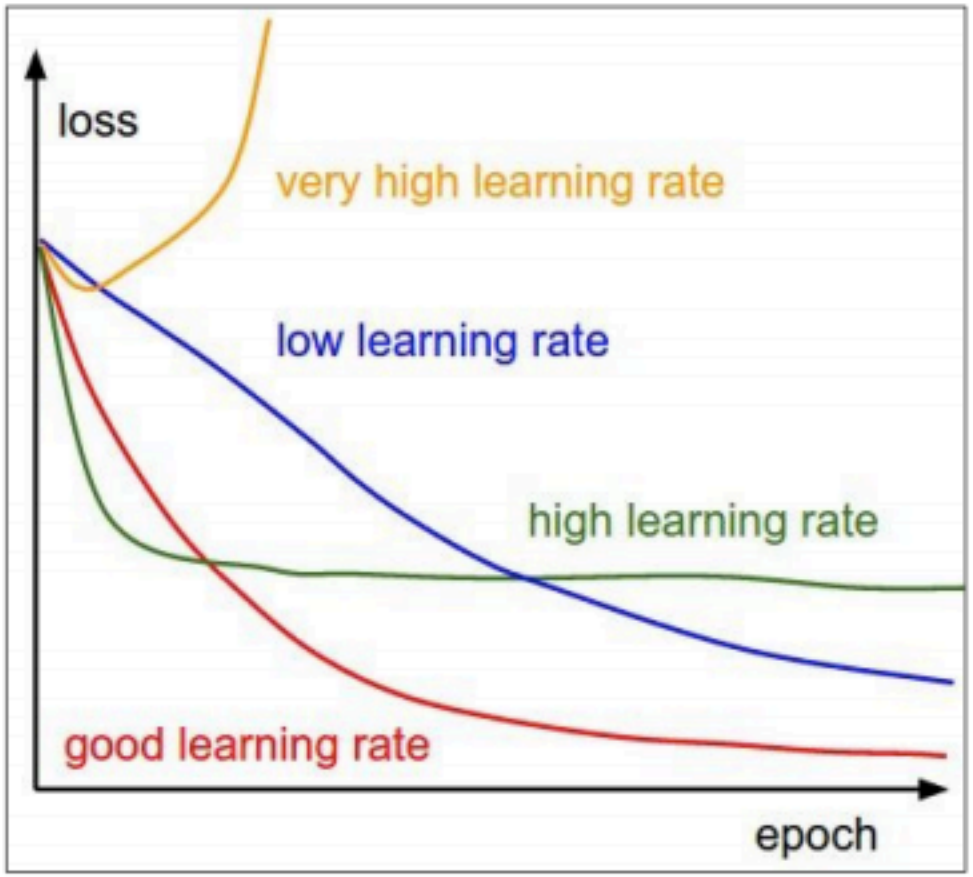
  ε: 10^{-8}

  } Usually these works well as default values and don't need to be tuned

Adam: adaptive moment estimation

Learning Rate Decay

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.

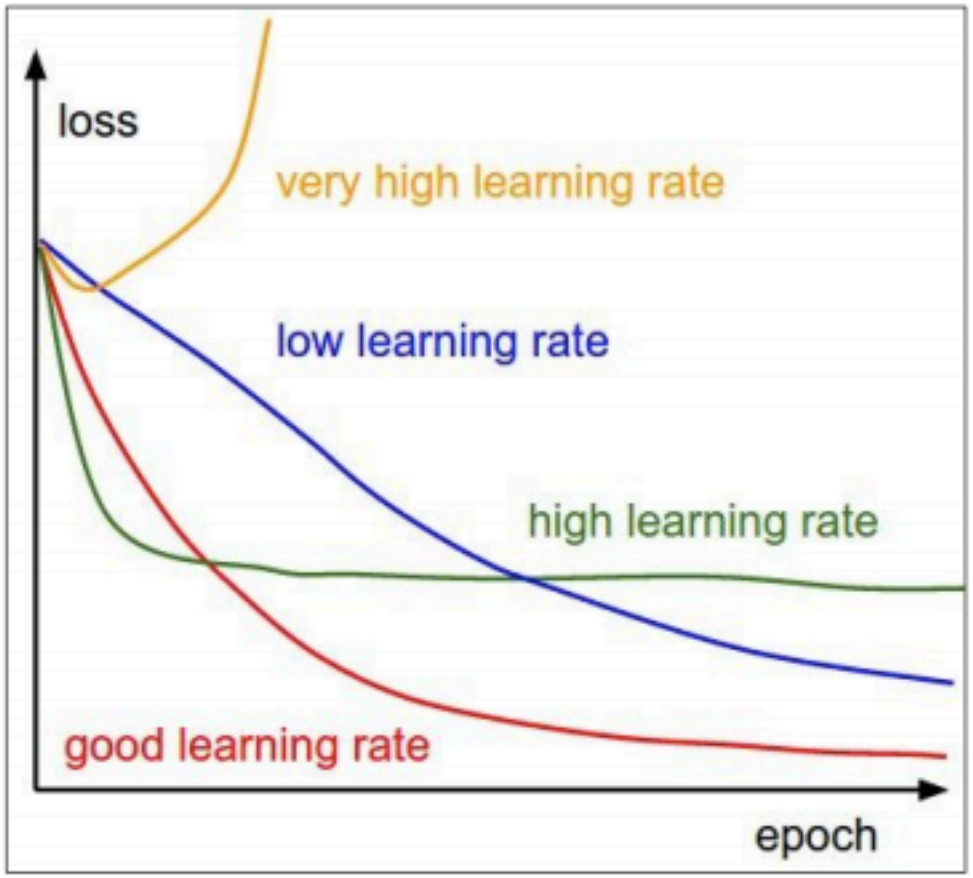
exponential decay:

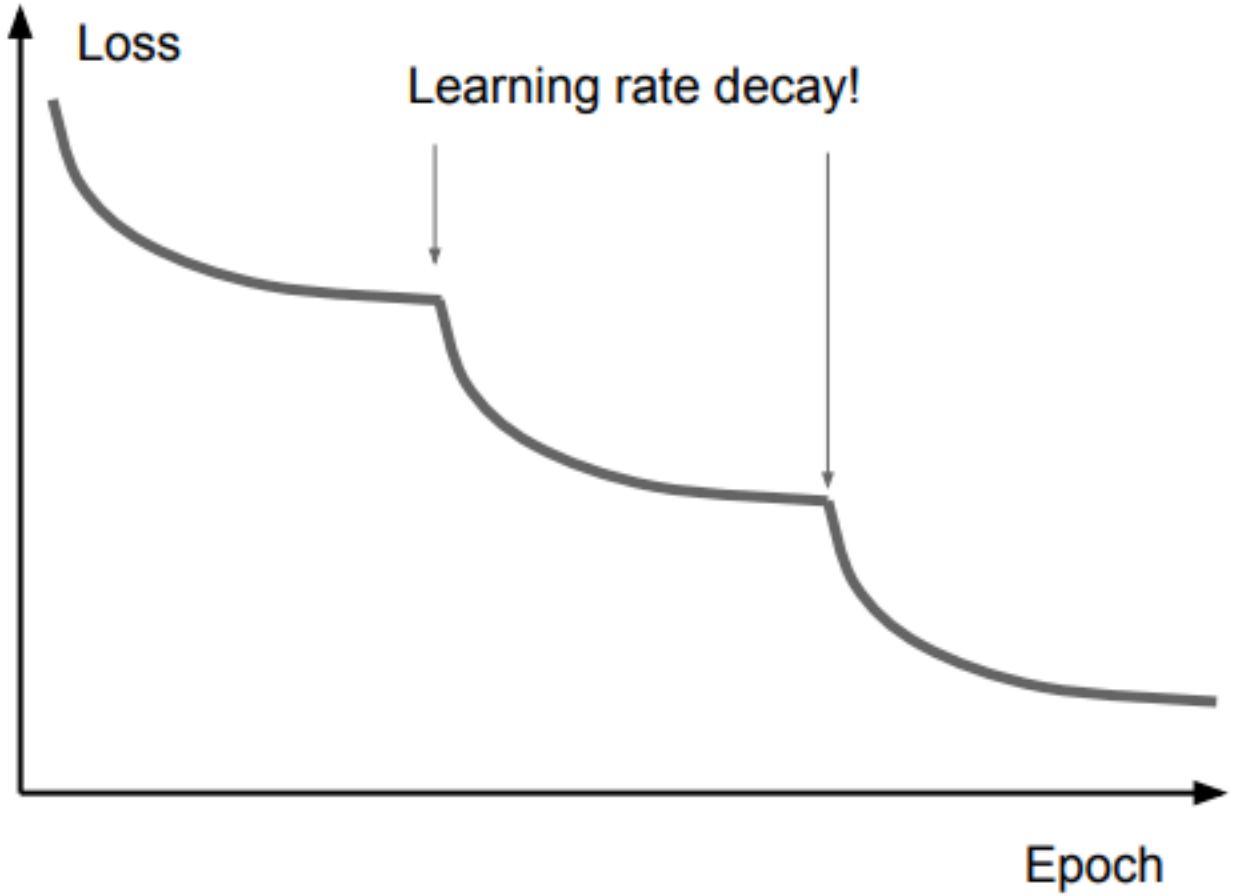
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

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

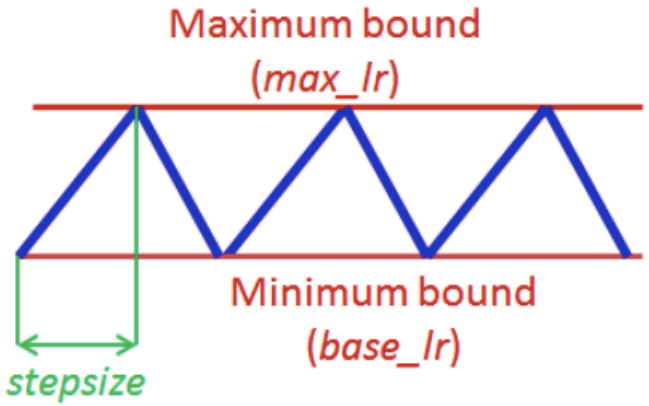
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.





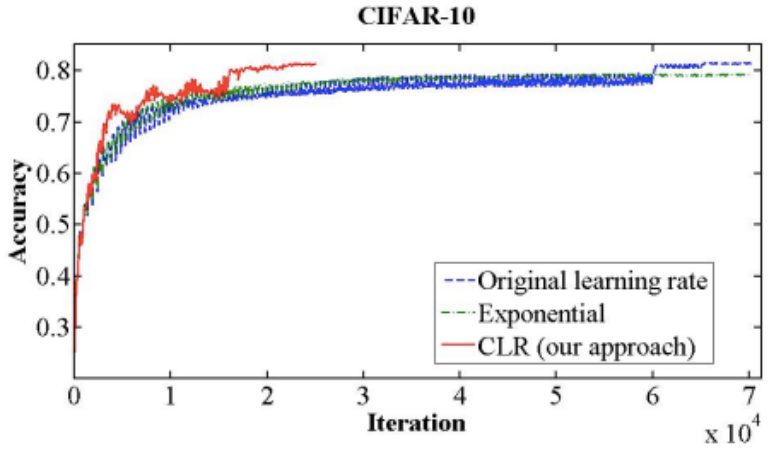
Cyclical Learning Rate Scheduler

- The intuition of CLR arises from the idea of letting the learning rate vary within a range of values rather than adopting a linearly or exponentially decreasing value.



Why CLR works?

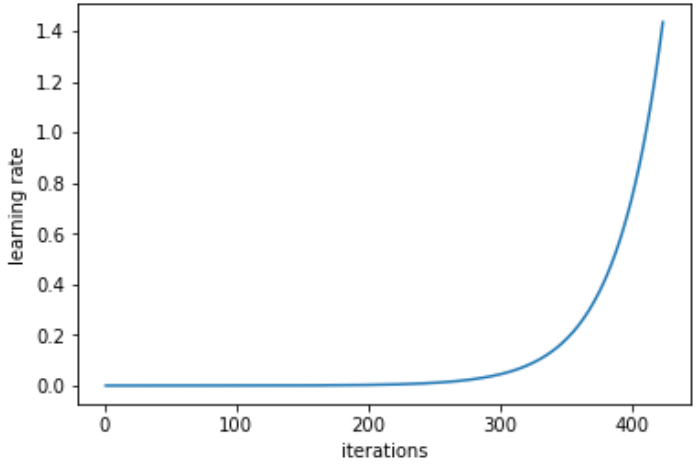
- The difficulty in minimizing the loss arises from sadd points rather than poor local minima. Saddle points have small gradients that slow the learning process. However, increasing the learning rate allows more rapid traversal of saddle point plateaus

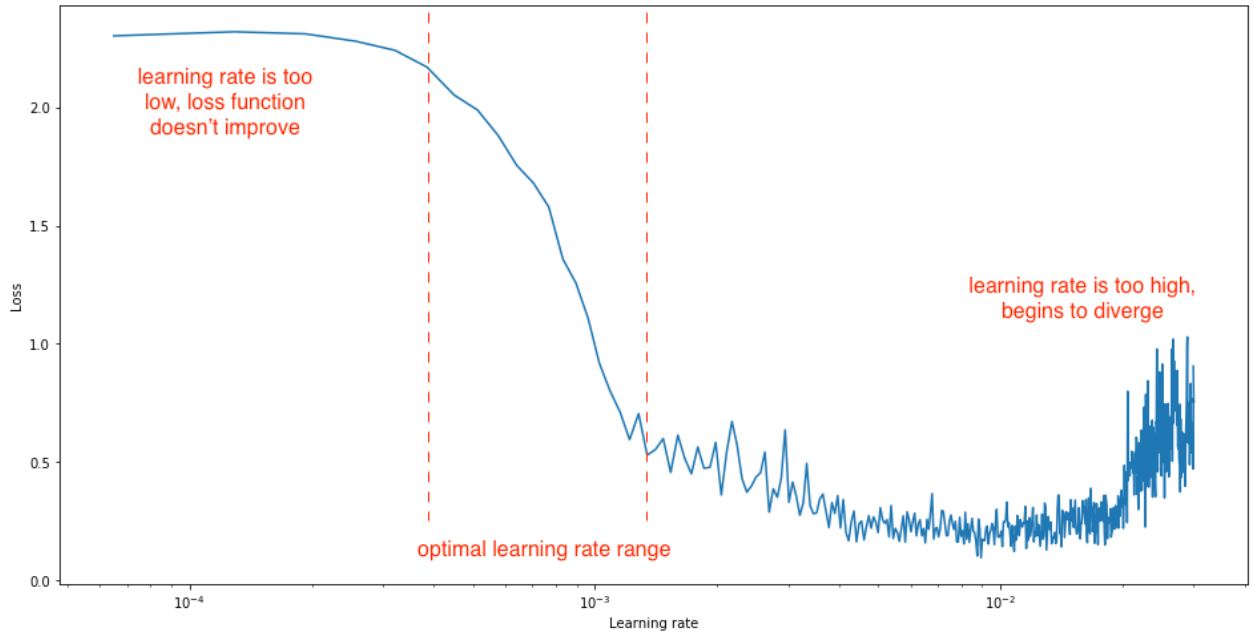


Cyclical Learning Rate Scheduler

How to find optimum range?

- The basic idea is to train a network starting from a low learning rate and increase the learning rate exponentially for every batch.

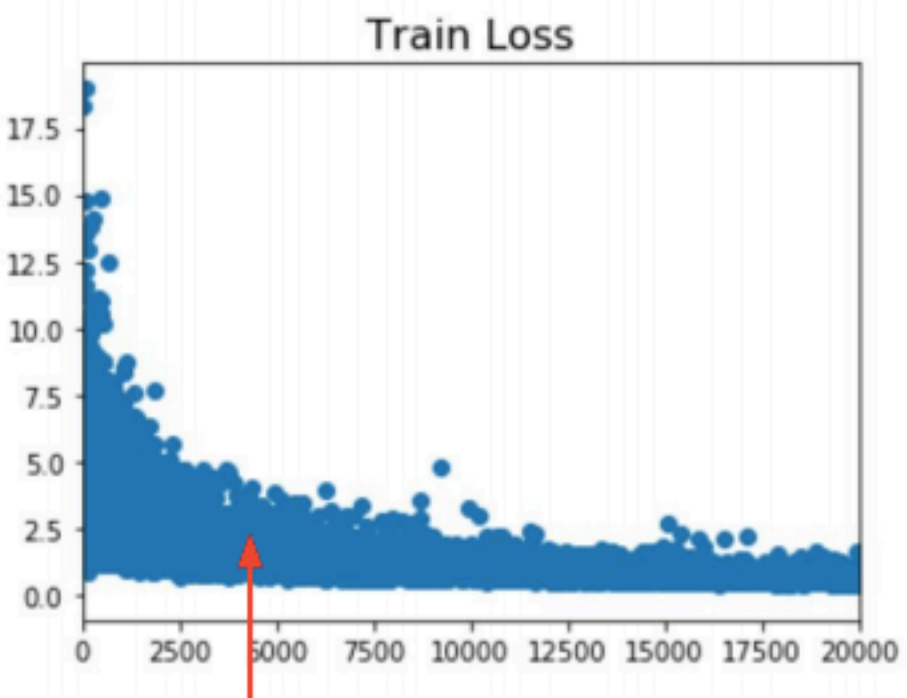


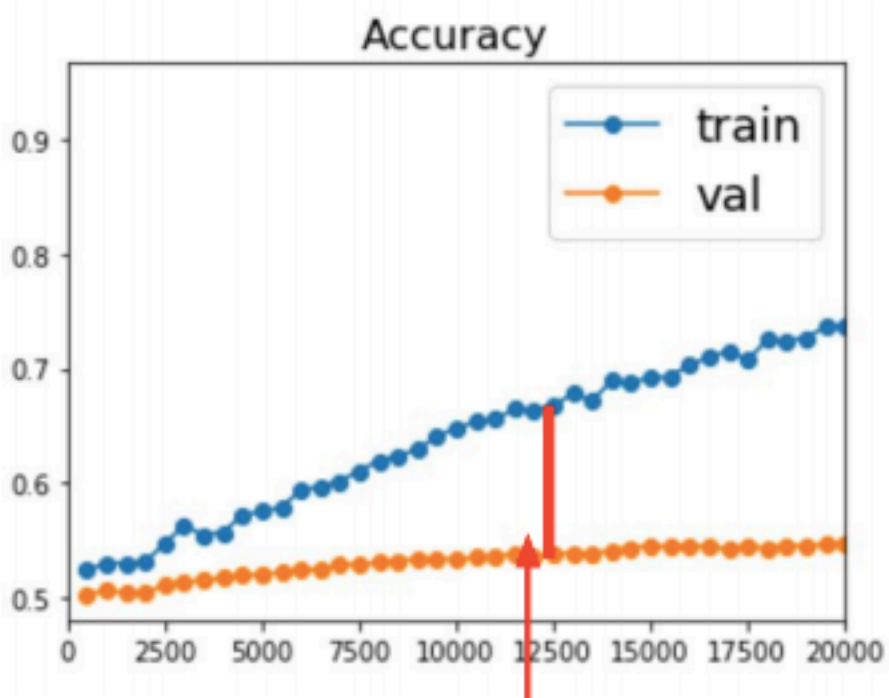


- The learning rate should be set within the range where the occurrence of loss decreases drastically.

Regularization

Beyond Training Error



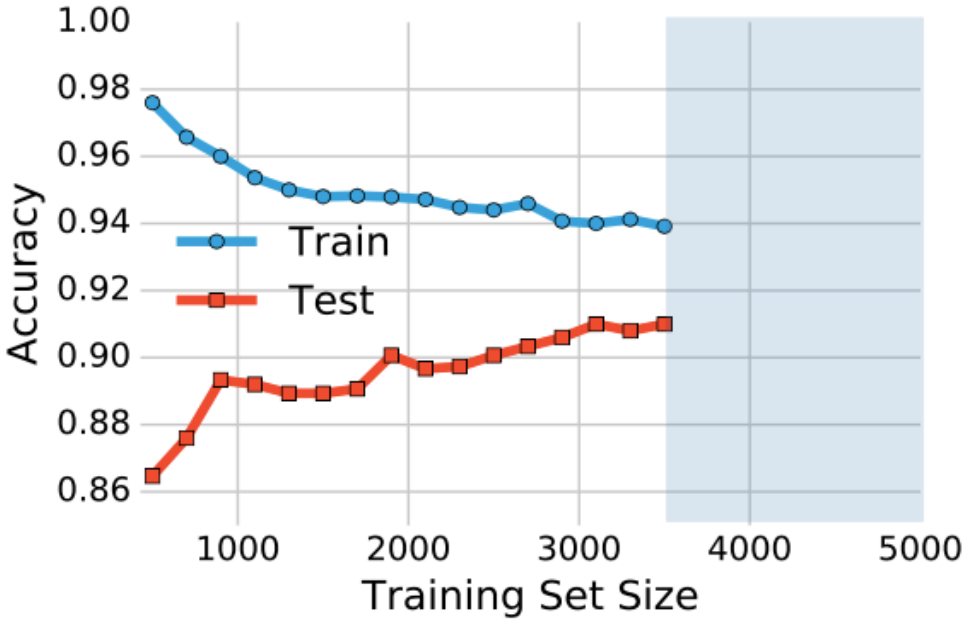


Better optimization algorithms help reduce training loss

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

Regularization: More Data

More Data

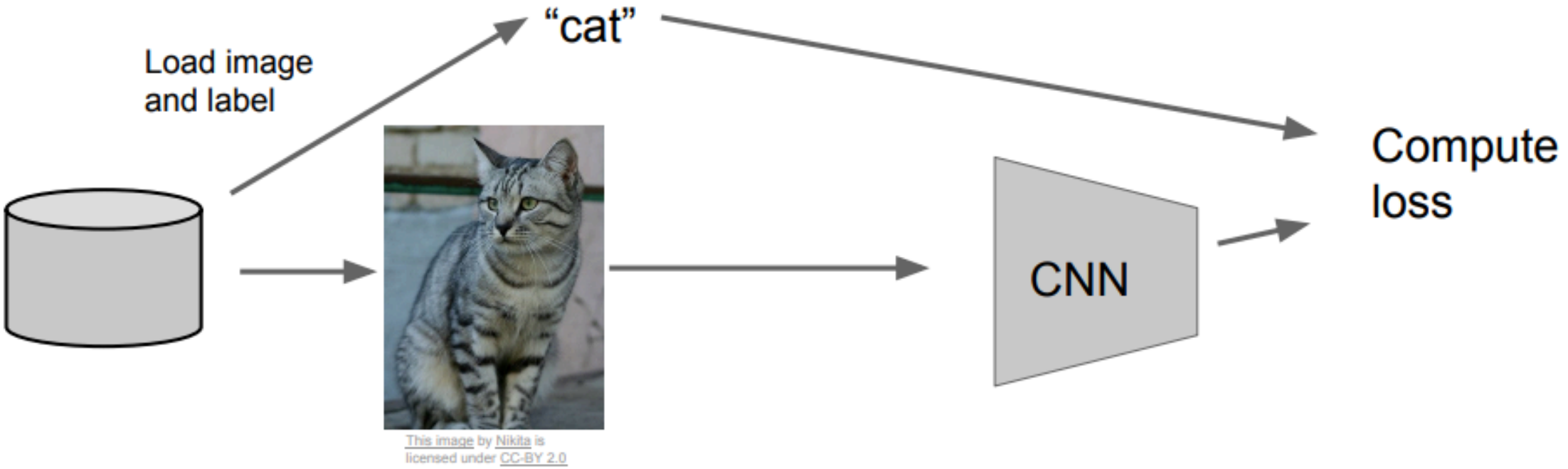


Softmax on MNIST subset (kept test set size constant)

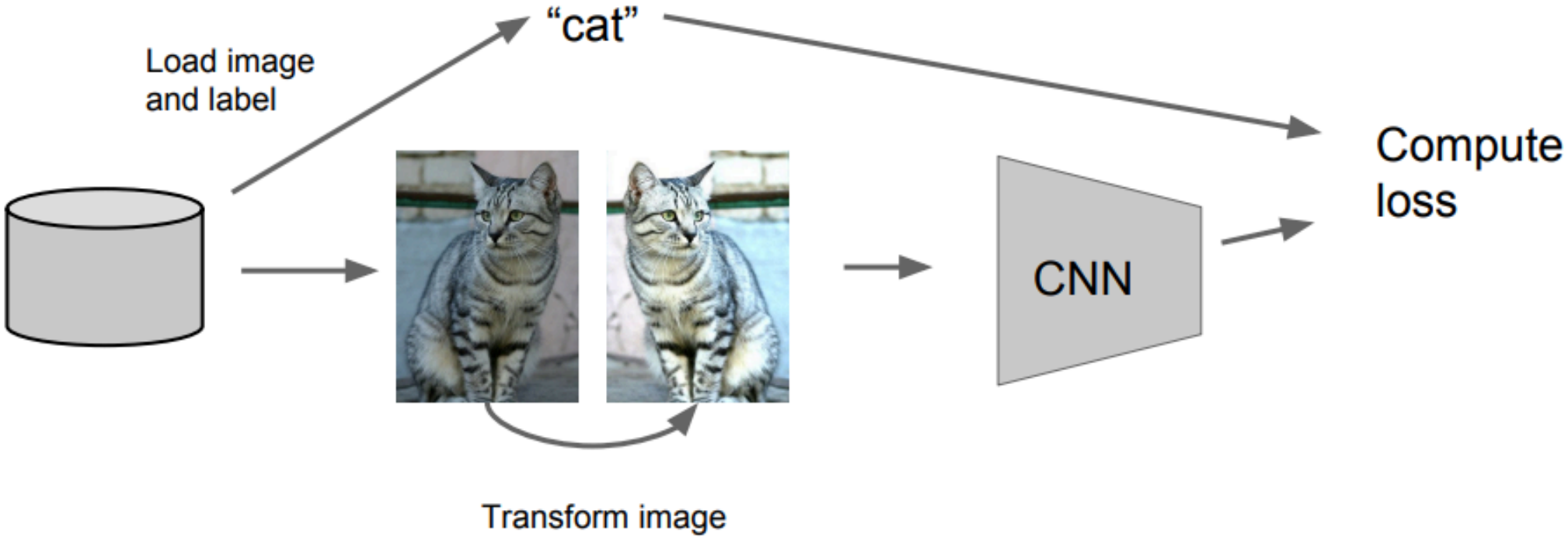
More Data

- Getting more data always works
- Data can be generated using augmentation
  - For images: random crop, rotation, flipping, ...)
- Usually more attention should be paid to data than to the model. Simple algorithm with well prepared data works better than complex models with raw and few data ("garbage in garbage out").

Data Augmentation

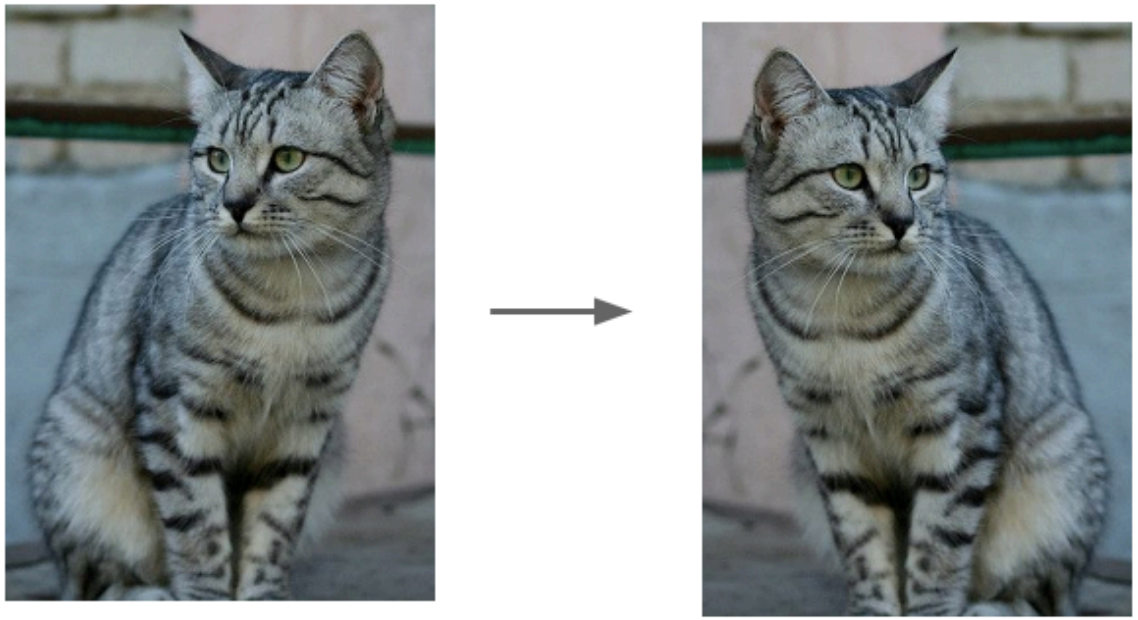


Data Augmentation



Data Augmentation

Horizontal Flips



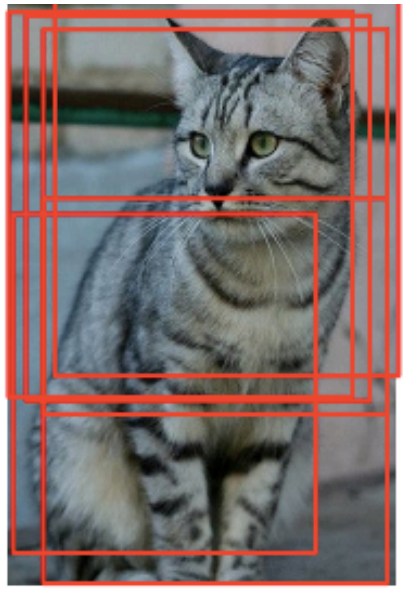
Data Augmentation

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



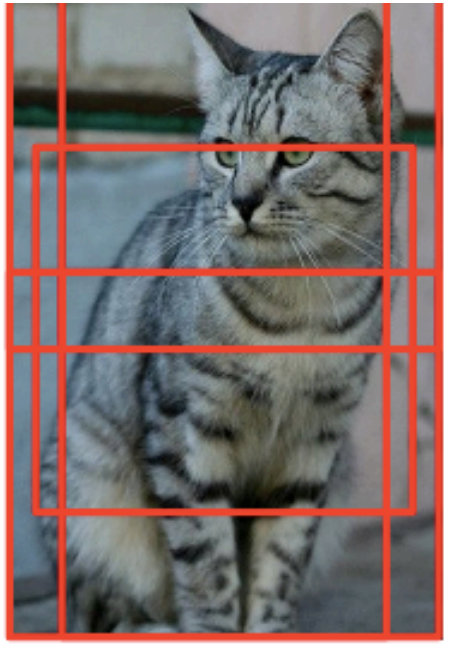
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, sh[illegible] cs231n_2018_lecture07.pdf
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 - Regularization

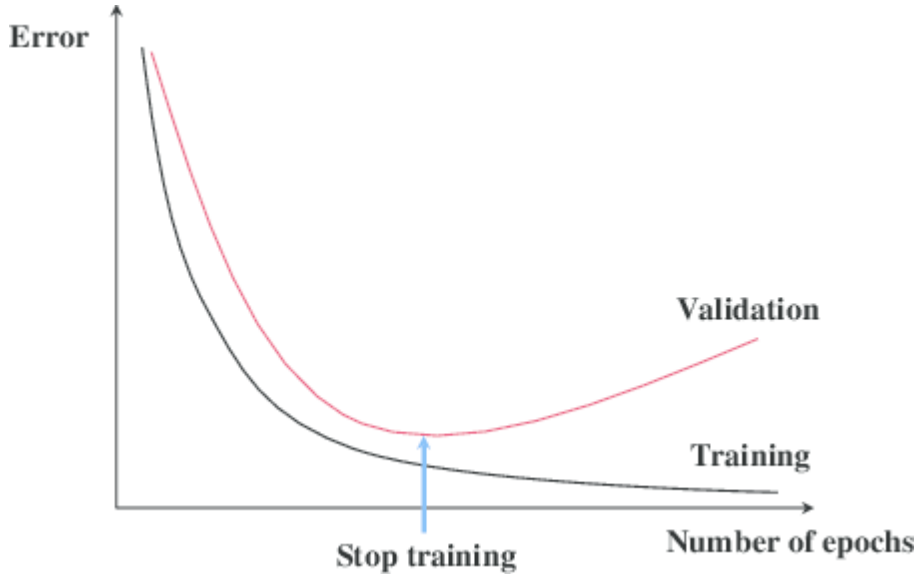
Data augmentation is a regularization as well because of random noise added while training.

Data Augmentation in PyTorch

```
training_transforms = torchvision.transforms.Compose([
    #torchvision.transforms.RandomRotation(degrees=20),
    #torchvision.transforms.Resize(size=(34, 34)),
    #torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomAffine(degrees=(-20, 20), translate=(0.15, 0.15),
                                        resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
])

# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html

train_dataset = datasets.MNIST(root='data',
                               train=True,
                               transform=training_transforms,
                               download=True)

test_dataset = datasets.MNIST(root='data',
                              train=False,
                              transform=test_transforms)
```

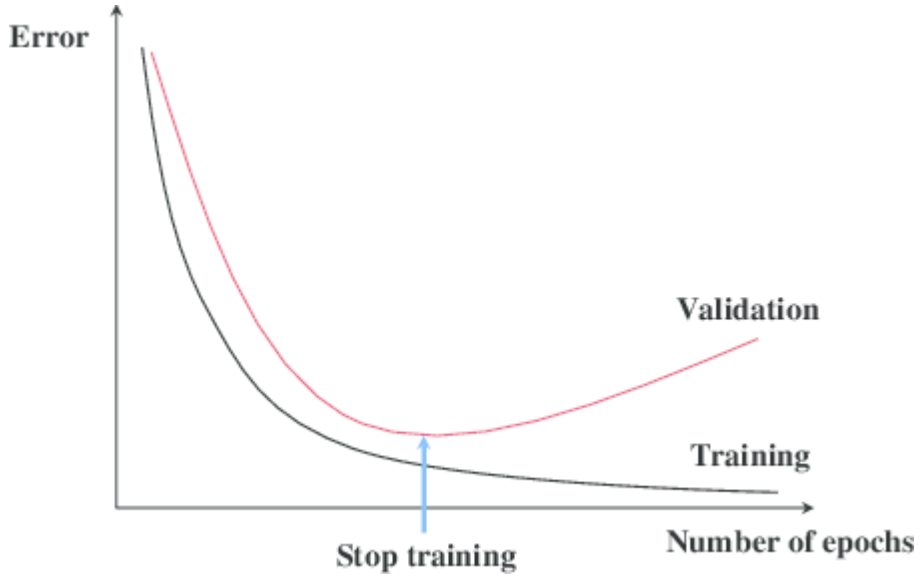
Regularization: Early Stopping

Early Stopping



Not very common anymore. Any ideas why?

Early Stopping



Orthogonalization:
- Optimize cost function
  - Gradient descent, momentum, Adam, ...
- Not overfit:
  - Regularization, getting more data

2 completely separate tasks

Not very common anymore. Any ideas why?

Regularization: Model Ensembles

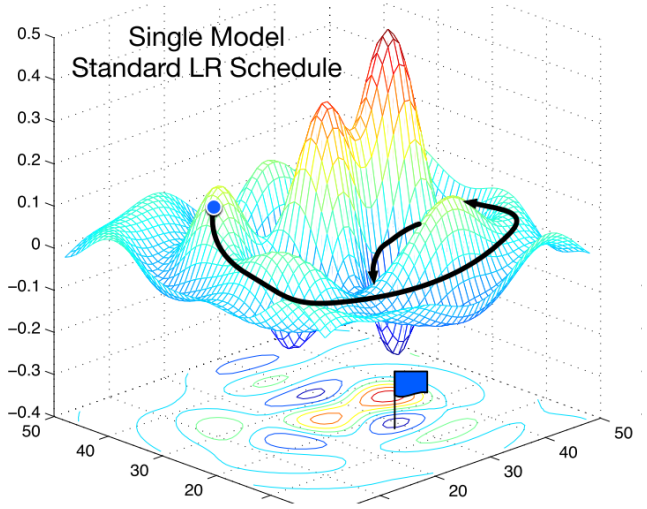
Model Ensembles

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

Enjoy 2% extra performance

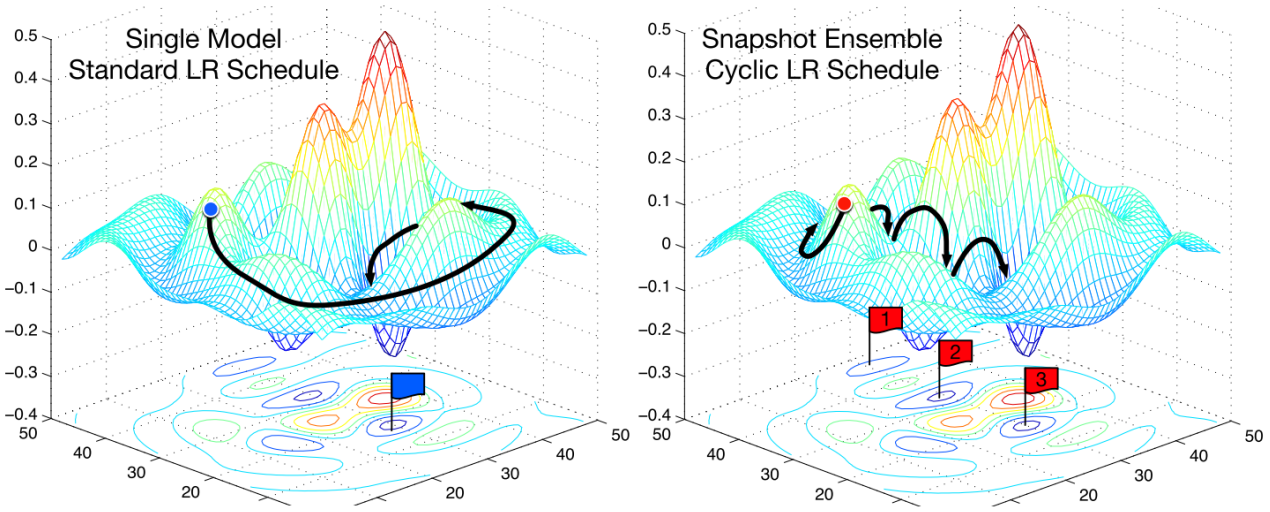
Snapshot Ensembles

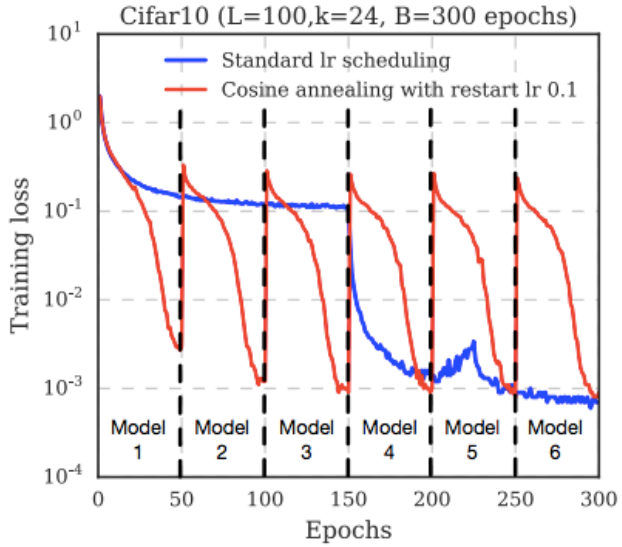
- Instead of training independent models, use multiple snapshots of a single during training



Snapshot Ensembles

- Instead of training independent models, use multiple snapshots of a single during training





Cyclical learning rate schedules can make this work even better

Regularization: Add Constraints Loss

L1/L2 Regularization

- L1 regularization => LASSO regression
- L2 regularization => Ridge regressioin

Basically, a “weight srhinkage” or a “penalty against complexity”

L1/L2 Regularization

$$\text{Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j} w_j^2$$

where: $\sum_{j} w_j^2 = ||\mathbf{w}||_2^2$

and λ is a hyperparameter

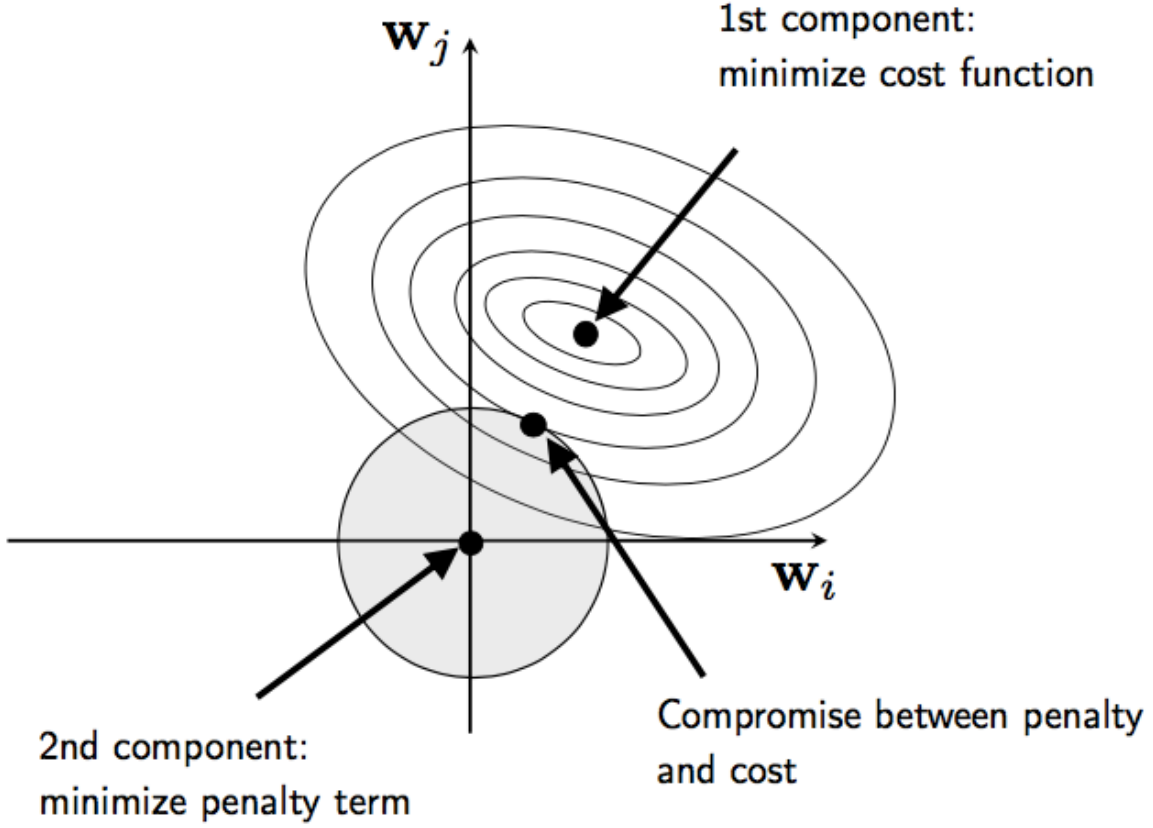
L1/L2 Regularization

$$\text{L1-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j}|w_j|$$

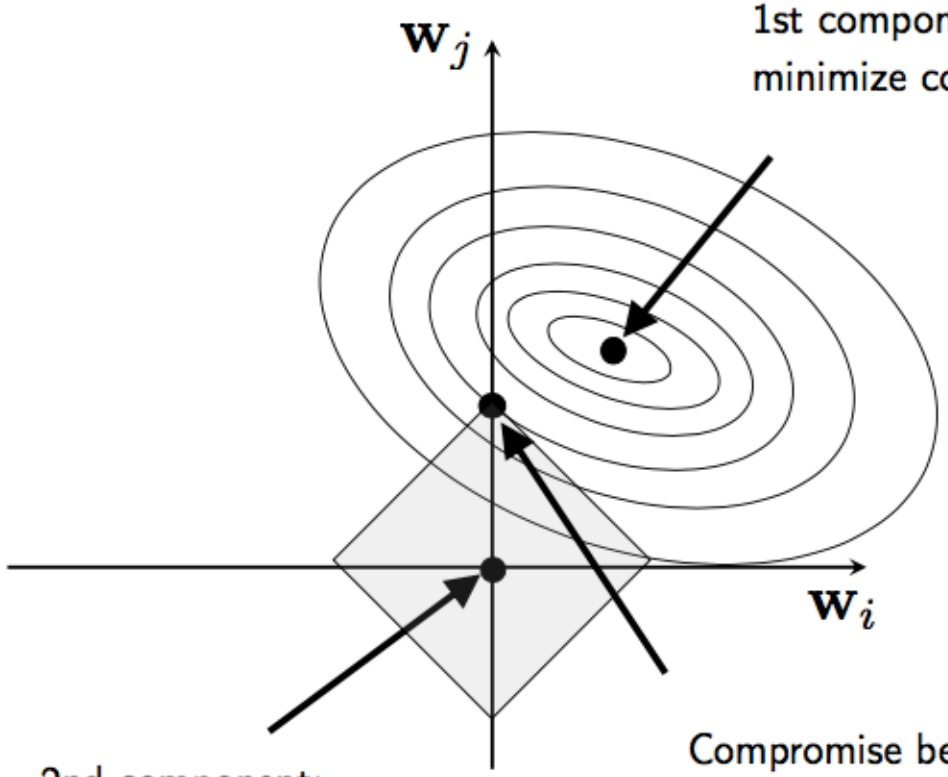
where: $\sum_{j}|w_j| = ||\mathbf{w}||_1$

- L1-regularization encourages sparsity (which may be useful)
- However, usually L1 regularization does not work well in practice and is very rarely used
- Also, it's not smooth and harder to optimize

L2 Regularization Visualization



L1 Regularization Visualization



L2 Regularization for Neural Nets

$$\text{L2-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{l=1}^{L}||\mathbf{w}^{(l)}||_F^2$$

sum over layers

where $||\mathbf{w}^{(l)}||_F^2$ is the Frobenius norm (squared):

$$||\mathbf{w}^{(l)}||_F^2 = \sum_i \sum_j (w_{i,j}^{(l)})^2$$

L2 Regularization for Neural Nets

Regular gradient descent update:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Gradient descent update with L2 regularization:

$$w_{i,j} := w_{i,j} - \eta \left(\frac{\partial \mathcal{L}}{\partial w_{i,j}} + \frac{2\lambda}{n} w_{i,j} \right)$$

L2 Regularization in PyTorch

Manually:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(num_epochs):
  #### Compute outputs ####
  out = model(X_train_tensor

  #### Compute gradients ####

  ##########################################################
  ## Apply L2 regularization (weight decay)
  cost = F.binary_cross_entropy(out, y_train_tensor, reduction='sum')
  cost = cost + 0.5 * LAMBDA * torch.mm(model.linear.weight,
                                        model.linear.weight.t())
  # note that PyTorch also regularizes the bias, hence, if we want
  # to reproduce the behavior of SGD's "weight_decay" param, we have to add
  # the bias term as well:
  cost = cost + 0.5 * LAMBDA * model.linear.bias**2

  optimizer.zero_grad()
  cost.backward()
```

L2 Regularization in PyTorch

Automatically:

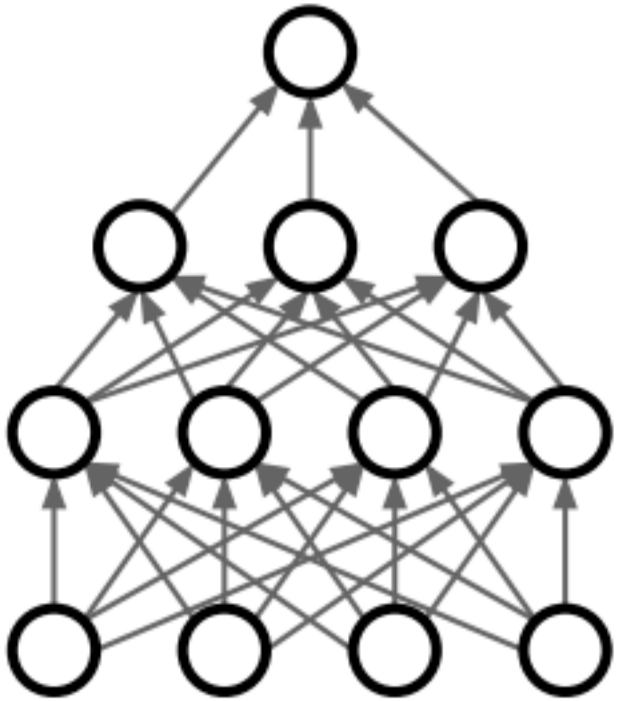
```
################################################################
## Apply L2 regularization
optimizer = torch.optim.SGD(model.parameters(),
                            lr=0.1,
                            weight_decay=LAMBDA)
#----------------------------------------------------------

for epoch in range(num_epochs):

  #### Compute outputs ####
  out = model(X_train_tensor)

  #### Compute gradients ####
  cost = F.binary_cross_entropy(out, y_train_tensor, reduction='sum')
  optimizer.zero_grad()
  cost.backward()
```

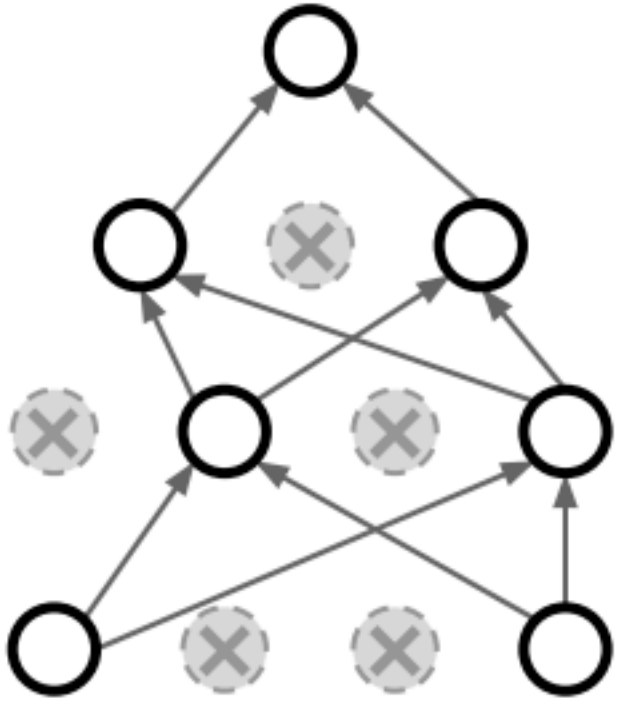
Dropout

Regularization: Dropout

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common

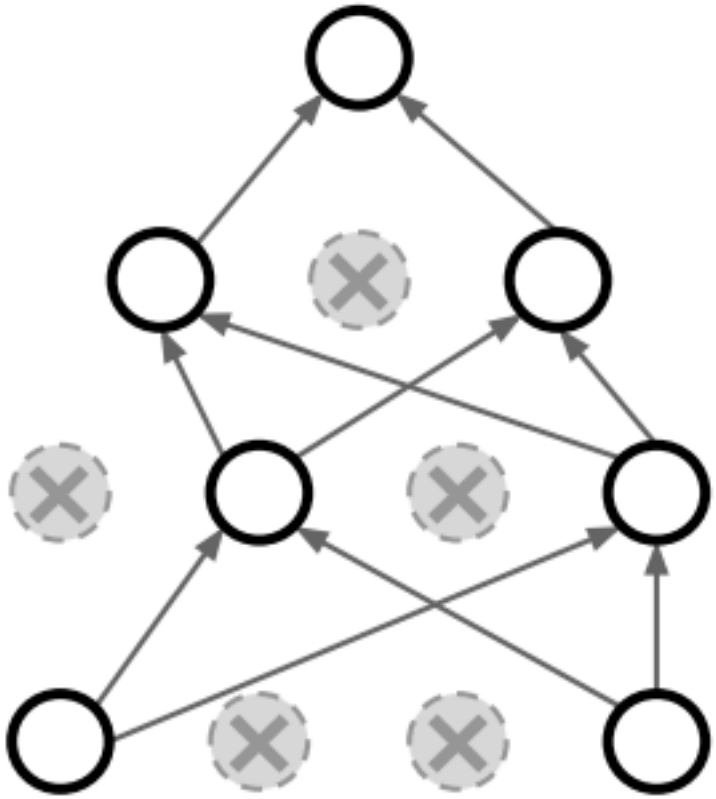




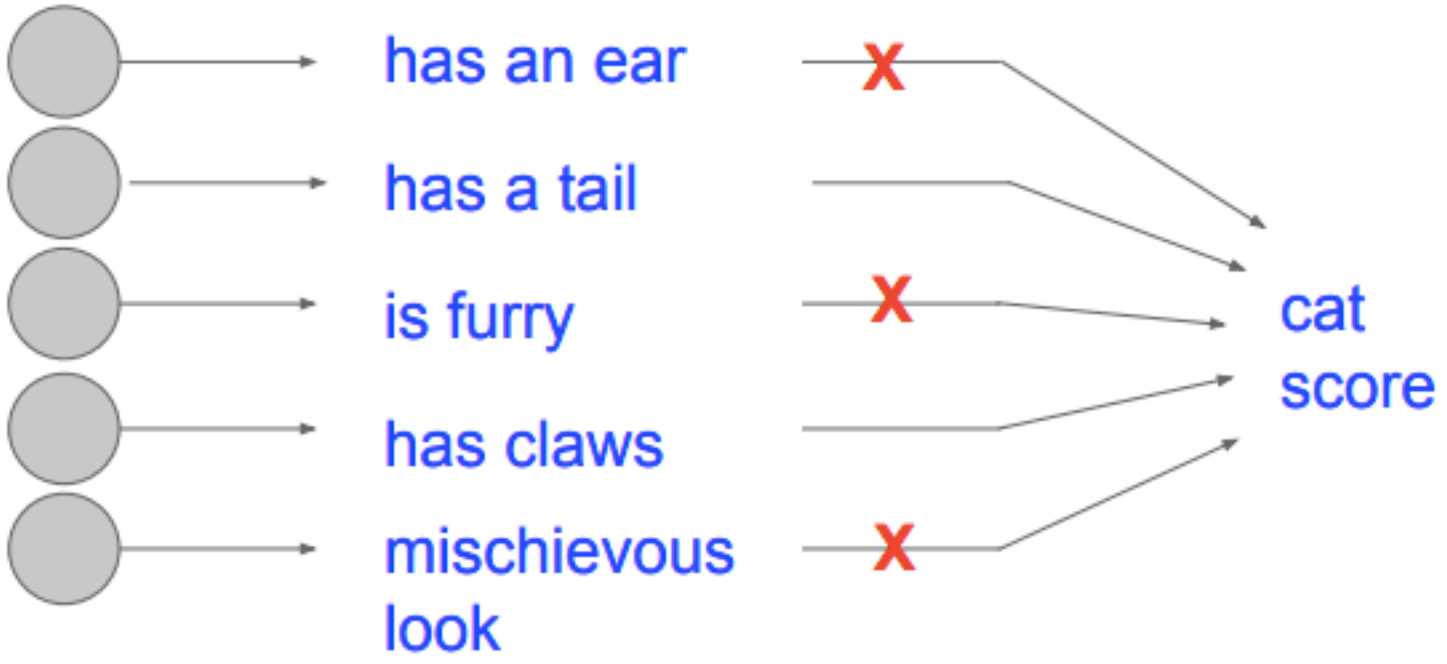
Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

Regularization: Dropout

How can this possibly be a good idea?

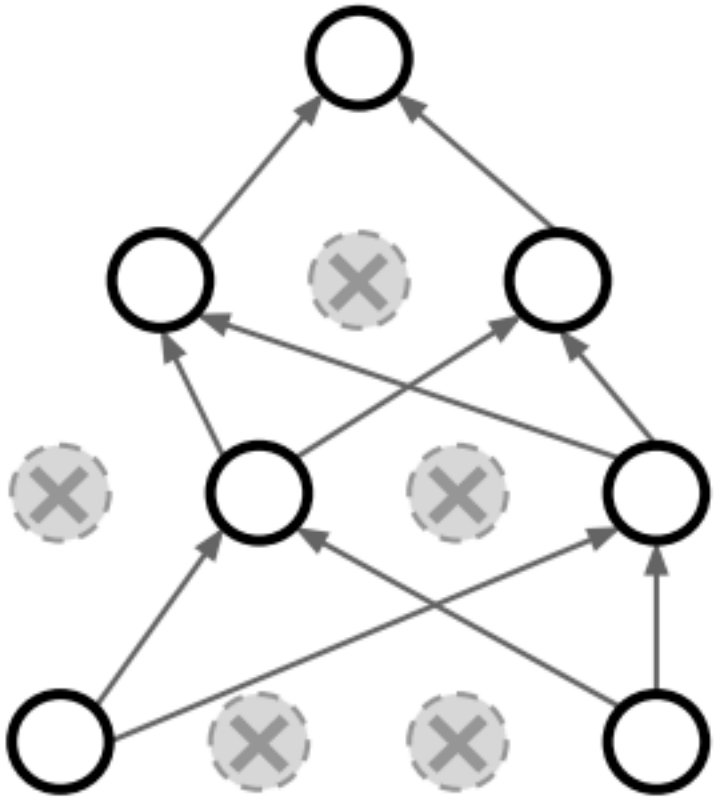


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

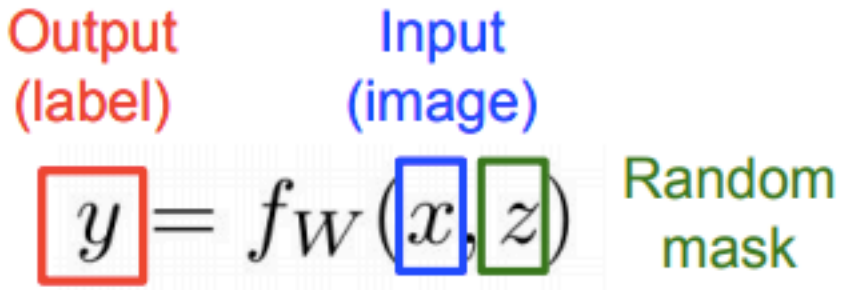
Dropout is training a large **ensemble** of models (that share parameters).

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...

Dropout: Test time

Dropout makes our output random!



Want to "average out" the randomness at test-time

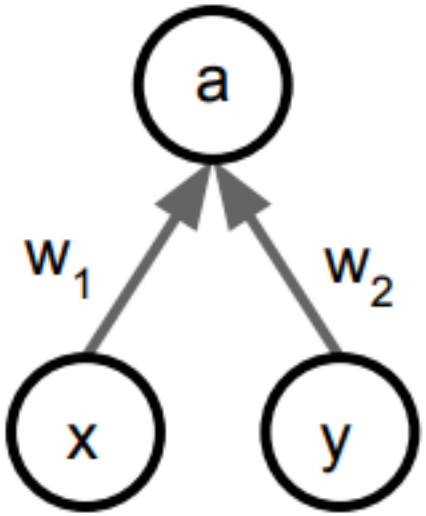
$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

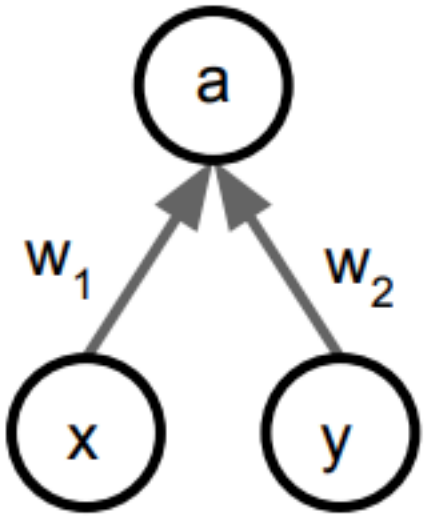
During training we have:

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout: Test time

Want to approximate the integral

$$y = f(x) = E_z\left[f(x,z)\right] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\left[a\right] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E\left[a\right] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

Dropout: Test time

```
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

Dropout Summary

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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)

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
```

drop in forward pass

scale at test time

More common: “Inverted dropout”

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

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

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

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

cs231n_2018_lecture07.pdf

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

Example: Batch Normalization

Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Dropout in PyTorch

Here, is is very important that you use `model.train()` and `model.eval()`!

```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)

        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()
        minibatch_cost.append(cost)
        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        cost = compute_loss(model, train_loader)
        epoch_cost.append(cost)
        print('Epoch: %03d/%03d Train Cost: %.4f' % (
                epoch+1, NUM_EPOCHS, cost))
        print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))
```

Dropout in PyTorch (OOP API)

```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

Dropout in PyTorch (Functional API)

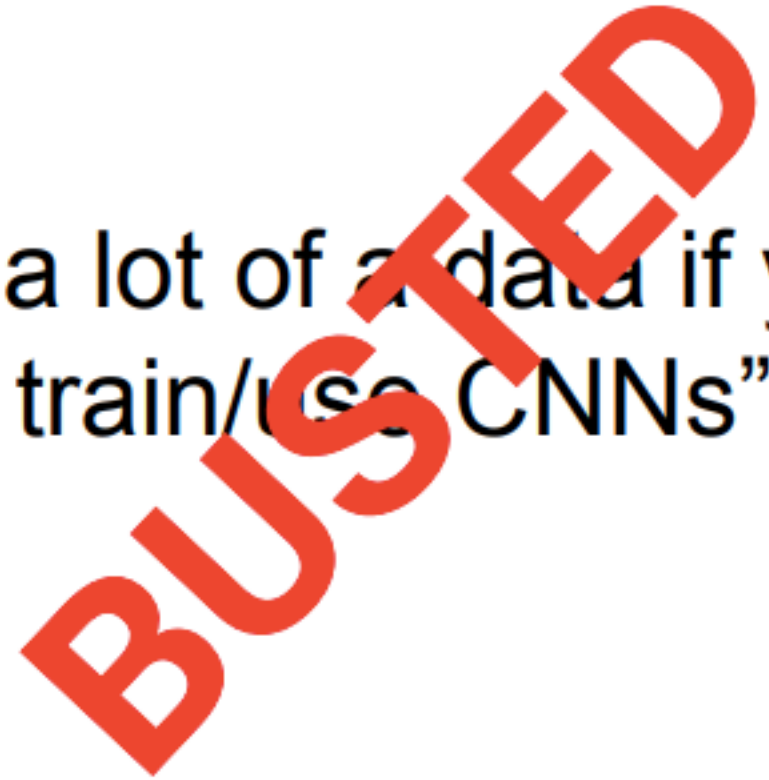
```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.drop_proba = drop_proba
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)

        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)

        self.linear_out = torch.nn.Linear(num_hidden_2,
                                          num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = F.dropout(out, p=self.drop_proba, training=self.training)
        out = self.linear_2(out)
        out = F.relu(out)
        out = F.dropout(out, p=self.drop_proba, training=self.training)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

Transfer Learning

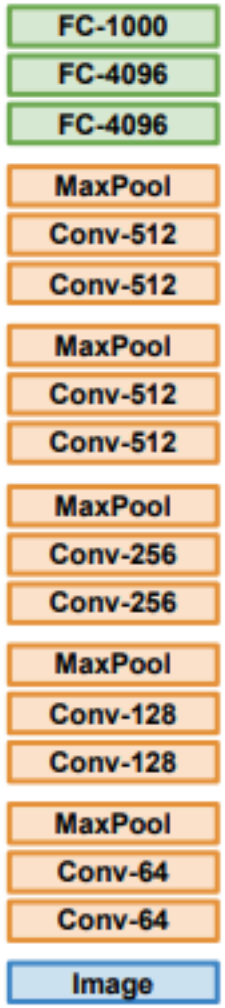
"You need a lot of a data if you want to train/use CNNs"



Transfer Learning with CNNs

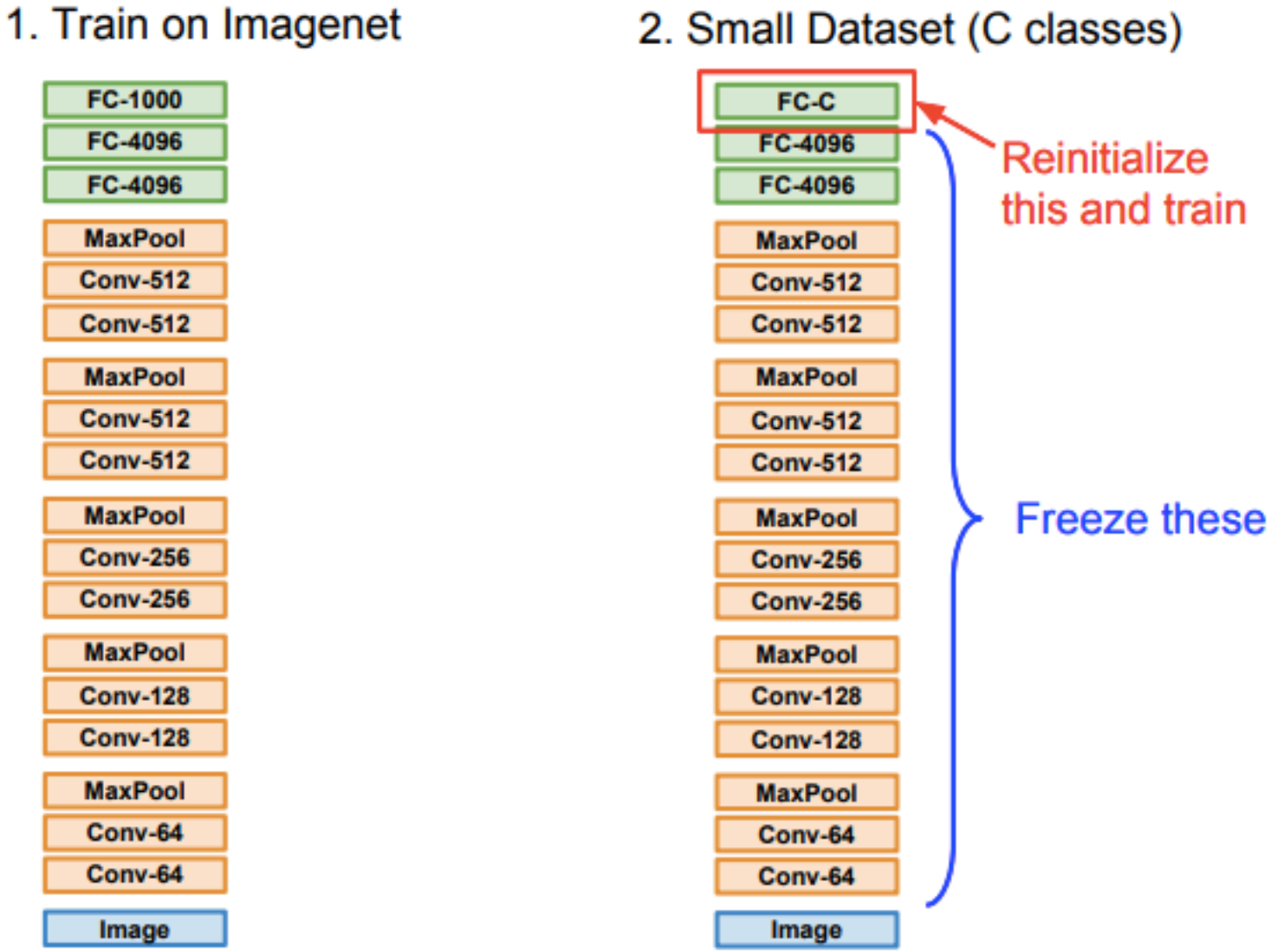
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



Transfer Learning with CNNs

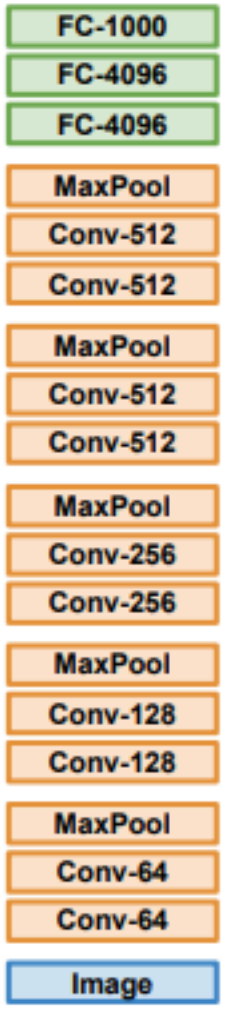
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



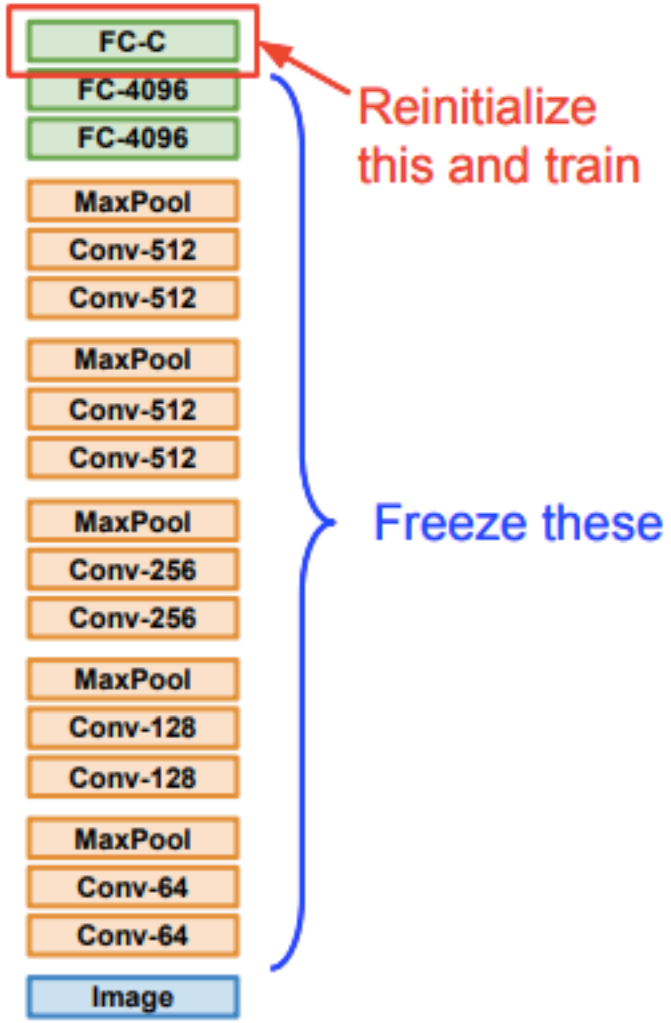
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

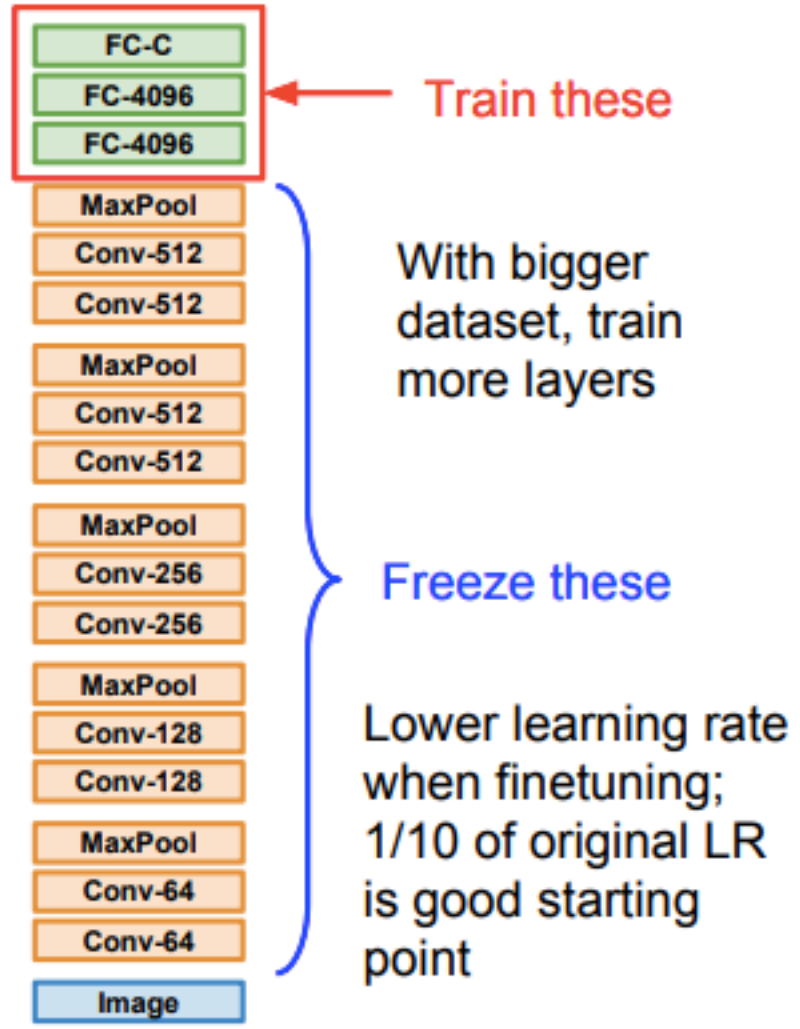
1. Train on Imagenet

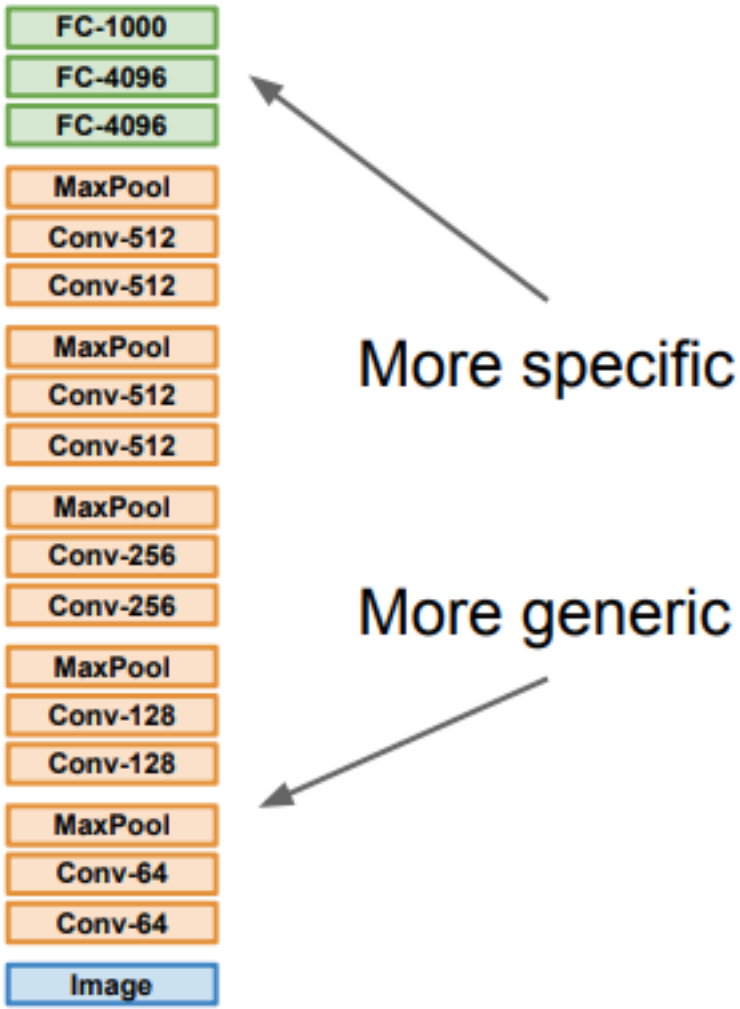


2. Small Dataset (C classes)

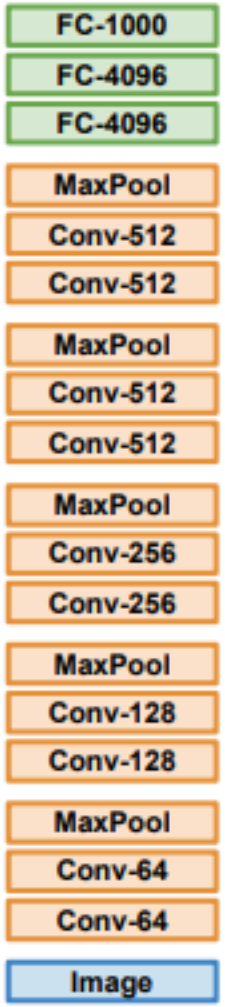


3. Bigger dataset

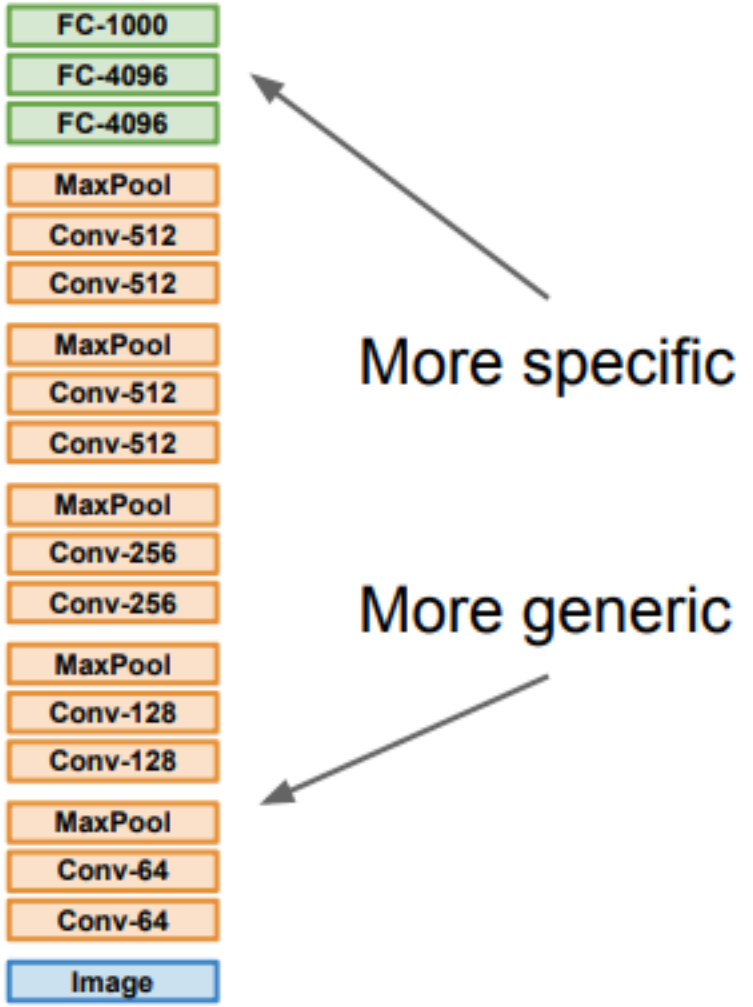




	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Resources Used

- Deeplearning.ai; Improving Deep Neural Networks: Hyperparameter Tuning; Andrew Ng
- CS231n Convolutional Neural Networks for Visual Recognition by Fei-Fei Li, Justin Johnson, Seran Yeung
- STAT 479: Deep Learning by Sebastian Raschka