USC Viterbi
School of Engineering

TRAINING DEEP NEURAL NETWORKS II

EE 541 – UNIT 7B

DR. BRANDON FRANZKE

Fall 2025

TOPIC OUTLINE

- Universal Approximation Theorem
  - Why Deep?
- A Gentle Introduction to PyTorch
- Vanishing gradient and activations
- Weight initialization
- Cost functions, regularization, dropout
- Optimizers
- Batch Normalization
- Hyperparameter optimization

COST (LOSS) FUNCTIONS

COST (LOSS) FUNCTIONS

some already covered, but let's review and see how they translate to PyTorch

simplified notation:

$\mathbf{s}$	last layer pre-activation (linear activation)
$\mathbf{a} = \mathbf{h}(\mathbf{s})$	last layer activation
$\mathbf{y}$	labels

Assume M output nodes, so these are $\boldsymbol{M} \times \mathbf{1}$ **vectors**

LOSS FUNCTIONS — L2 FOR REGRESSION

$$C = \|\mathbf{y} - \mathbf{a}\|_2^2 = \sum_{i=1}^{M} (y_i - a_i)^2$$

(squared) L2 norm of error
or sum of squared error

$$C = \frac{1}{M}\|\mathbf{y} - \mathbf{a}\|_2^2 = \frac{1}{M}\sum_{i=1}^{M} (y_i - a_i)^2$$

average squared error

these are equivalent

PyTorch implements the mean by default, see options
(good since it is normalized for number of classes)

```
ms = nn.MSELoss()
output = ms(
    torch.FloatTensor([[1, 1, 1], [2, 2, 2]]),
    torch.FloatTensor([[0, 0, 0], [3, 3, 3]]))
```

```
tensor(1.)
```

for BP Initialization

$$\frac{d}{da}(y - a)^2 = 2(y - a)$$

LOSS FUNCTIONS — L1 FOR REGRESSION

$$C = \|\mathbf{y} - \mathbf{a}\|_1 = \sum_{i=1}^{M} |y_i - a_i|$$

L1 norm of error
or sum of absolute error

$$C = \frac{1}{M}\|\mathbf{y} - \mathbf{a}\|_1 = \frac{1}{M}\sum_{i=1}^{M} |y_i - a_i|$$

average absolute error

these are equivalent

PyTorch implements the mean by default, see options
(good since it is normalized for number of classes)

```
ms = nn.L1Loss()
output = ms(
    torch.tensor([[1., 1., 1.], [1., 1., 1.]]),
    torch.tensor([[0., 0., 0.], [3., 3., 3.]]))
```

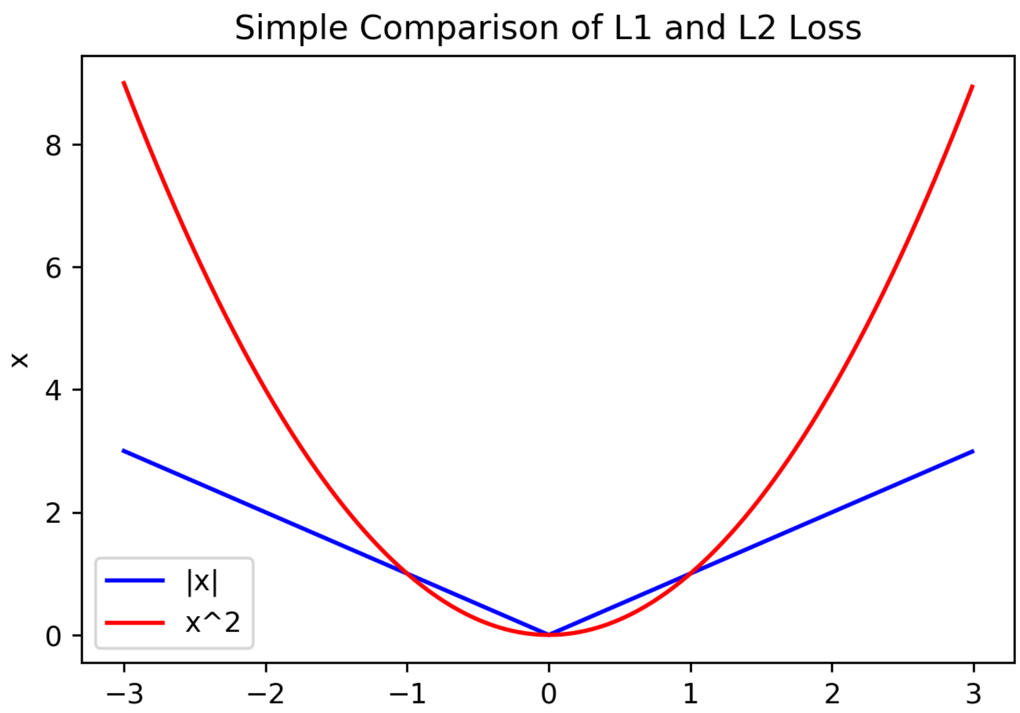
```
tensor(1.5000)
```

for BP Initialization

$$\frac{d}{da}|y - a| = \mathrm{sgn}(y - a)$$

$$= \begin{cases} +1 & a > y \\ -1 & a < y \end{cases}$$

LOSS FUNCTIONS — L1 VS L2



L2 penalizes large error more than L1

L2 corresponds to power/energy for ECE

L1 will typically induce sparsity in your weights - allows a few large weights and many other weights are near 0

LOSS FUNCTIONS — MULTICATEGORY CROSS ENTROPY

$$C = -\sum_{i=1}^{M} y_i \ln a_i = \sum_{i=1}^{M} y_i \ln\left(\frac{1}{a_i}\right)$$

BP gradient initialization: $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}$

If activations are outputs of a <u>softmax</u> then interpret as <u>probability of class i</u>

LOSS FUNCTIONS — MULTICATEGORY CROSS ENTROPY

Recall, with one-hot (hard labels)

$$C = -\sum_{i=1}^{M} y_i \ln a_i \longrightarrow C = -\ln a_m$$

Class m is true

```
loss_func = nn.CrossEntropyLoss()
loss_func(
    torch.tensor([
        [np.log(0.9),  np.log(0.05), np.log(0.05)],
        [np.log(0.05), np.log(0.89), np.log(0.06)],
        [np.log(0.05), np.log(0.01), np.log(0.94)]]
    ),
    torch.tensor([0,1,2])
).numpy()
```

```
array(0.09458991)
```

```
(np.log(0.9) + np.log(0.89) + np.log(0.94)) / 3
= -0.094590
```

(averaged over batch size)

Recall, MCE is the negative log-likelihood (NLL) with regression error model:

$$P(class = i) = a_i$$

LOSS FUNCTIONS — MULTICATEGORY CROSS ENTROPY

With soft labels we use the general form

$$C = -\sum_{i=1}^{M} y_i \ln a_i$$

```python
def categorical_cross_entropy(y_pred, y_true):
    return -(y_true * torch.log(y_pred)).sum(dim=1).mean()


categorical_cross_entropy(
    torch.tensor([
        [0.9,  0.05, 0.05],
        [0.05, 0.89, 0.06],
        [0.05, 0.01, 0.94]]
    ),
    torch.tensor([
        [0.7,  0.2, 0.1],
        [0.05, 0.9, 0.05],
        [0.3, 0.3, 0.4]]
    )
)
```

```
tensor(1.2243)
```

PyTorch does not include soft-label loss function

Write your own (left) or use `nn.KLDivLoss`

Recall, KL-divergence is a constant offset from MCE between the y and a probability mass functions

COST (LOSS) FUNCTIONS — BINARY CROSS ENTROPY

for $M = 2$ outputs – binary classification

$$C = -y\ln(a) - (1-y)\ln(1-a) = y\ln\left(\frac{1}{a}\right) + (1-y)\ln\left(\frac{1}{1-a}\right)$$

Same as MCE with $a_0 = a,\ a_1 = 1 - a$

PyTorch uses this

```
nn.BCELoss()(
    torch.tensor([[.6, .8, .1]]),
    torch.tensor([[0., 1., 0.]])
)
```

```
tensor(0.4149)
```

```
def bce(y,a):
    return -1*y*np.log(a+1e-10) -(1-y)*np.log(1-a+1e-10)

np.mean(bce(np.array([0,1,0]), np.array([0.6, 0.8, 0.1])))
0.414932
```

Compare with `nn.BCEWithLogitsLoss()`

CROSS ENTROPY LOSS — "FROM LOGITS"

numerically simpler (and <u>more stable</u>) to compute

$\text{Loss}(\text{activation}(s))$ in one step

example: **binary cross entropy**

$$C = -y\ln(a) - (1-y)\ln(1-a)$$

$$a = \sigma(s)$$

$$= [1+e^{-s}]^{-1}$$

$$C = y\ln(1+e^{-s}) + (1-y)\ln(1+e^{+s})$$

$$= \ln\left(1+e^{\overline{y}s}\right)$$

$$\overline{y} = (-1)^{y}$$

$$C = \ln\left(1+e^{\overline{y}s}\right)$$

loss directly from linear activation

Use this if you do not need a pmf out of your trained model

– *i.e.*, if you will threshold the outputs of the trained model

Compare with `nn.NLLLoss()`

CROSS ENTROPY LOSS — "FROM LOGITS"

numerically simpler (and more stable) to compute

$\text{Loss}(\text{activation}(s))$ in one step

example: **multicategory cross entropy**

$$C = -\sum_{i=1}^{M} y_i \ln\left[\frac{e^{s_i}}{\sum_j e^{s_j}}\right]$$

$$= -\sum_{i=1}^{M} y_i[s_i - K(\mathbf{s})]$$

$$= -\sum_{i=1}^{M} y_i s_i + K(\mathbf{s})$$

$$K(\mathbf{s}) = \ln\left(\sum_j e^{s_j}\right)$$

$$C = K(\mathbf{s}) - \sum_{i=1}^{M} y_i s_i$$

loss directly from linear activation

$$C = K(\mathbf{s}) - s_m$$

Class m is true, hard labels

CROSS ENTROPY LOSS — "FROM LOGITS"

$$K(\mathbf{s}) = \ln\left(\sum_j e^{s_j}\right)$$
$$= \max_j^* s_j$$
$$\max^*(x, y) = \ln(e^x + e^y)$$
$$= \max(x, y) + \ln\left(1 + e^{-|x-y|}\right)$$
$$\max^*(x, y, z) = \ln(e^x + e^y + e^z)$$
$$= \max^*(\max^*(x, y), z)$$

numerically stable approach

loss directly from linear activation:

$$C = \max_j^* s_j - \sum_{i=1}^{M} y_i s_i$$

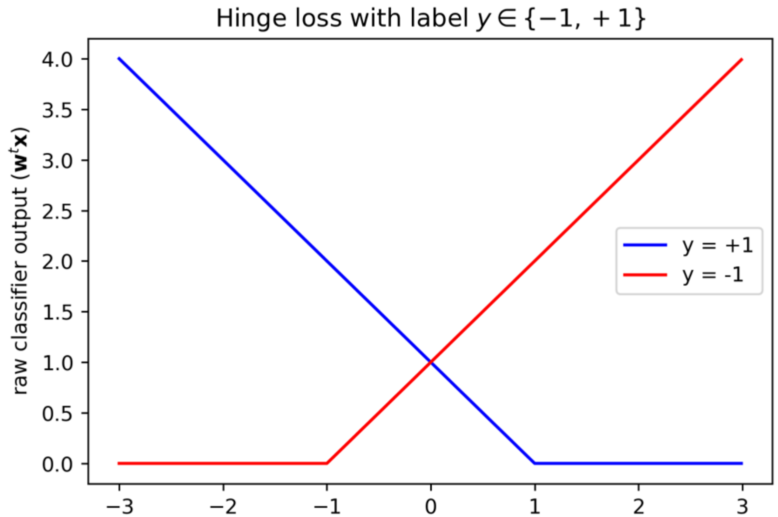
$$C = \max_j^* s_j - s_m$$

Class m is true, hard labels

HINGE LOSS

for binary classifier with target/labels in $\{-1, +1\}$

penalize misclassification (threshold)



$$C = \max(1 - ya, 0) \qquad a = s, y \in \{-1, +1\}$$

typically use linear output activation

`torch.nn.MarginRankingLoss, margin=1`

PYTORCH LOSS FUNCTIONS

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y .
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y .
`nn.CrossEntropyLoss`	This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.
`nn.CTCLoss`	The Connectionist Temporal Classification loss.
`nn.NLLLoss`	The negative log likelihood loss.
`nn.PoissonNLLLoss`	Negative log likelihood loss with Poisson distribution of target.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss measure
`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between the target and the output:
`nn.BCEWithLogitsLoss`	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
`nn.MarginRankingLoss`	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch *Tensors*, and a label 1D mini-batch tensor y (containing 1 or -1).

https://pytorch.org/docs/master/nn.html#loss-functions

CUSTOM LOSS FUNCTIONS

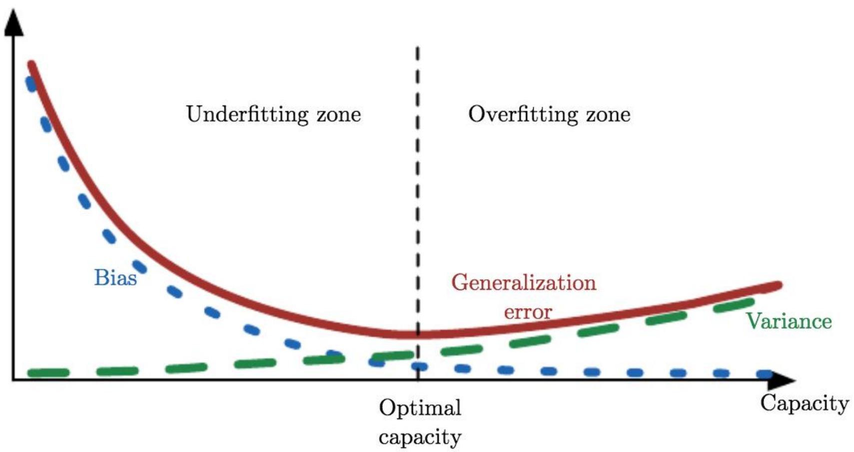
PyTorch = simple custom loss functions

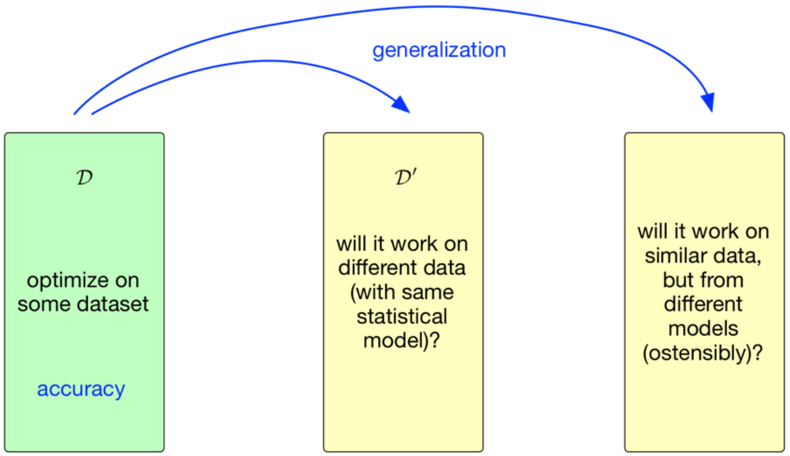
```
def my_loss(output, target):
    loss = torch.mean((output - target)**2)
    return loss

model = nn.Linear(2, 2)
x = torch.randn(1, 2)
target = torch.randn(1, 2)
output = model(x)
loss = my_loss(output, target)
loss.backward()
print(model.weight.grad)
```

reimplementation of `nn.MSELoss`

WEIGHT REGULARIZATION

WHY REGULARIZE





trade-off between over and under fitting
is the Bias-Variance trade-off

Main goal of
Machine Learning
is to GENERALIZE

I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016, http://www.deeplearningbook.org.

REGULARIZERS

Main goal of Machine Learning is to **GENERALIZE**

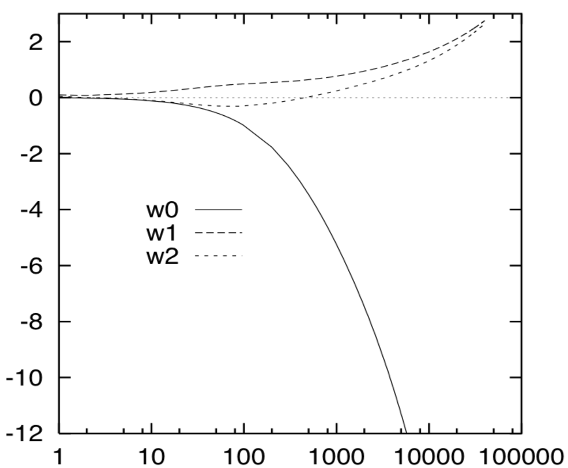
regularization is anything you do in training that is aimed at
improving generalization over accuracy —
i.e., anything that does not optimize the cost on the training data

When people say "regularizer" they usually mean a narrower definition:

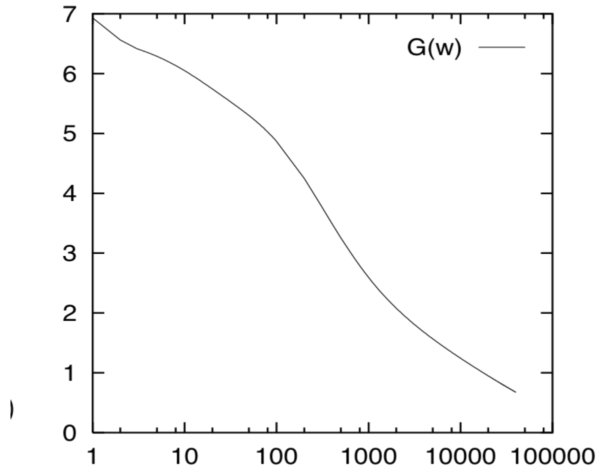
an additive term to the loss function that prevents weights from getting too large

HOW TO REGULARIZE

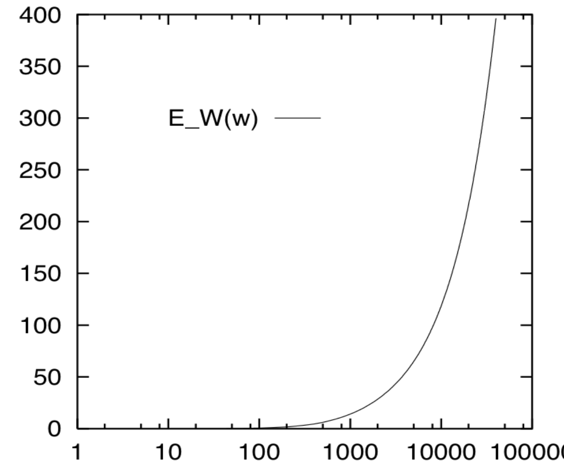
Why do large weights correspond to over-fitting???



weight evolution



learning curve (loss)



L2 norm of weights

MacKay, Information Theory and Inference, Cambridge University Press, 2003

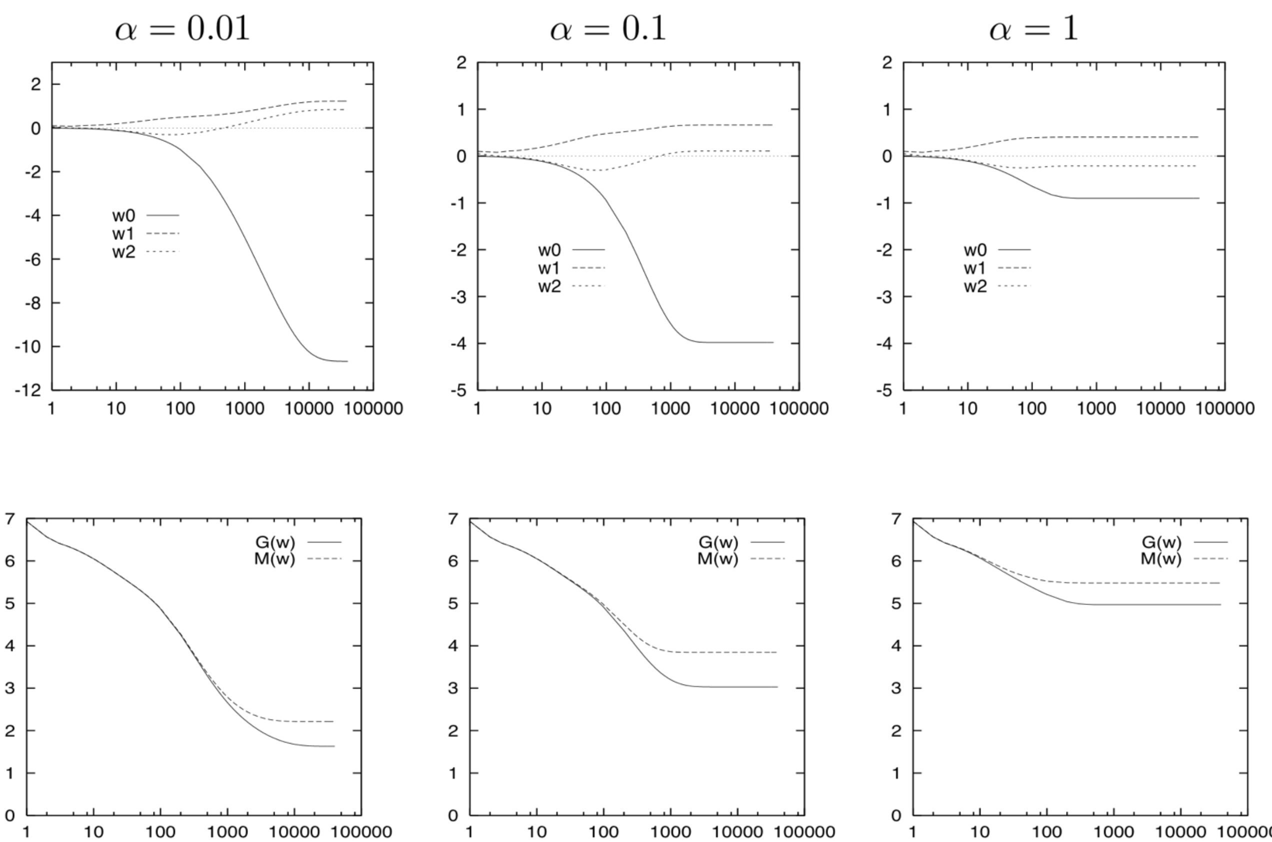
HOW TO REGULARIZE

This is an experimental observation

regularizer coefficient

weight evolution
(L2 regularization)

learning curve (loss)



MacKay, Information Theory and Inference, Cambridge University Press, 2003

REGULARIZERS — L1, L2

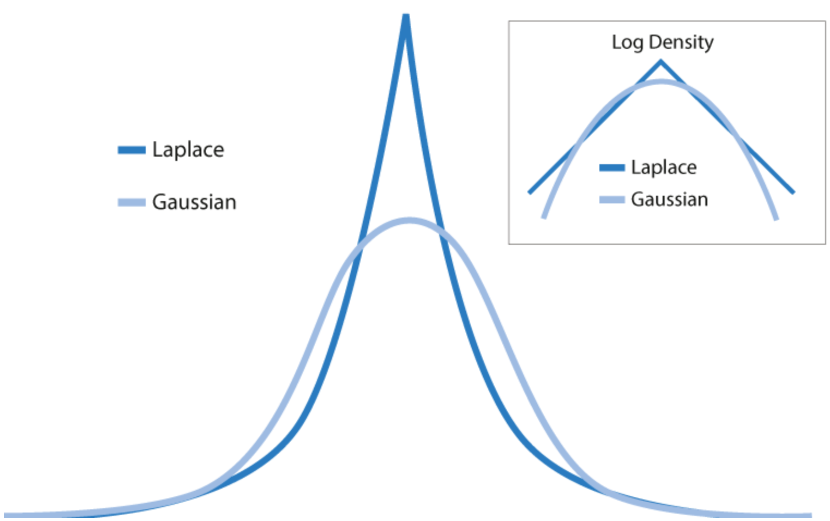
L2 regularization (***weight decay***)

$$C = C_{\text{no-reg}} + \lambda \|\mathbf{w}\|_2^2 \longrightarrow w \leftarrow w - \eta\left(\frac{\partial C}{\partial w} + 2\lambda w\right)$$

L1 regularization (***LASSO***)

$$C = C_{\text{no-reg}} + \lambda \, \|\mathbf{w}\|_1 \longrightarrow w \leftarrow w - \eta\left(\frac{\partial C}{\partial w} + \lambda \, \text{sgn}(w)\right)$$

As we saw earlier: these can be viewed as being induced by an *a priori* distribution on the weights with MAP weight estimation



L2: Gaussian prior

L1: Laplace prior

REGULARIZERS

$$\lambda \approx \frac{\text{Importance of small weights}}{\text{Importance of minimizing training loss}}$$

$\lambda = 0$ → $w^* \sim \arg\min C_{\text{no-reg}}(\mathbf{w})$ — could be over-fitting, depends on capacity of model, dataset properties, and inference problem

$\lambda = \infty$ → $w^* \sim 0$ — under-fitting

Typically: $10^{-5} \lesssim \lambda \lesssim 10^{-3}$

REGULARIZERS IN PYTORCH

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

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)

```
optim.SGD([
    {'params': model.base.parameters(), 'weight_decay': 0.},
    {'params': model.classifier.parameters(), 'weight_decay': 0.01}
], lr=1e-2)
```

Use per-parameter options for more control

Most optimizers include a **`weight_decay`** parameter

L^2 penalty, default = 0

works with autograd package

REGULARIZERS IN PYTORCH

But how to back-propagate with regularized loss???

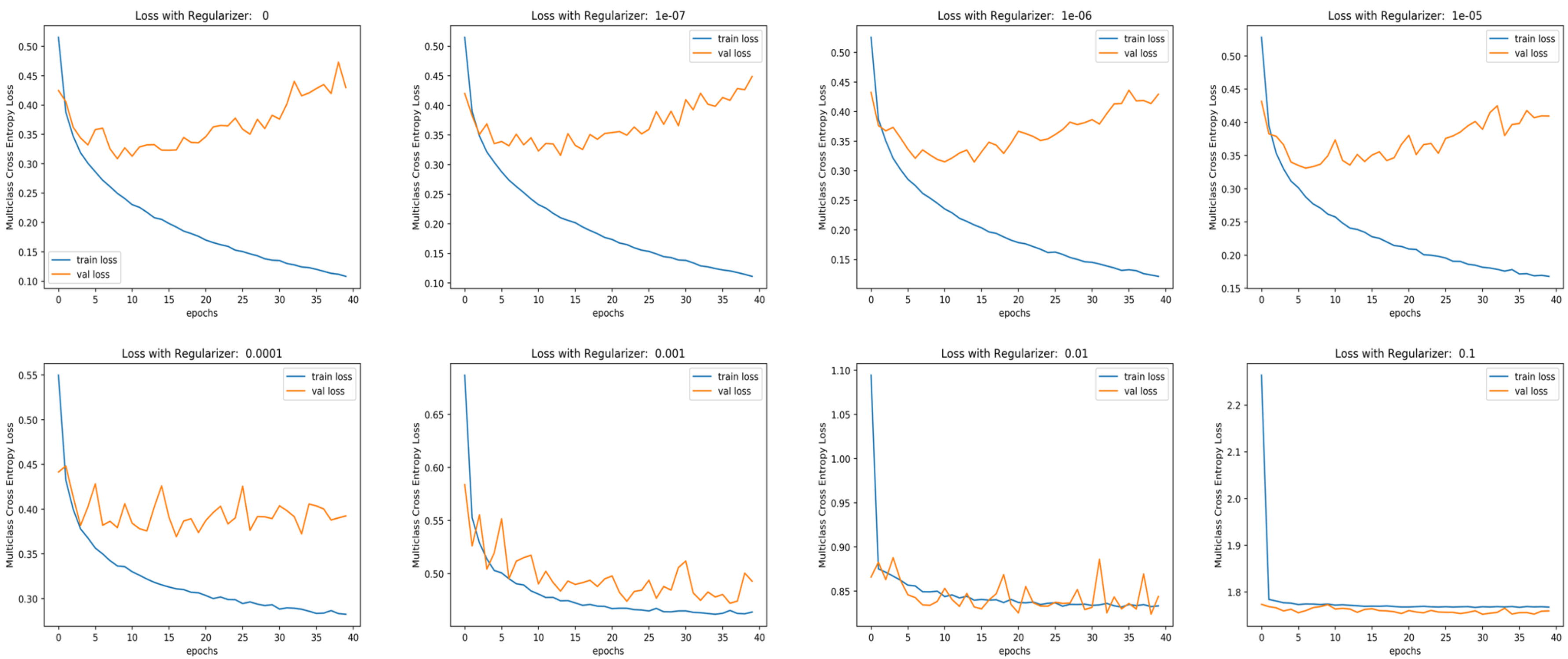
```python
l1_regularization, l2_regularization = torch.tensor(0), torch.tensor(0)

optimizer.zero_grad()
outputs = model(inputs)
cross_entropy_loss = F.cross_entropy(outputs, targets)

for param in model.parameters():
    l1_regularization += torch.norm(param, 1)
    l2_regularization += torch.norm(param, 2)**2

loss = cross_entropy_loss + l1_regularization + l2_regularization
loss.backward()
optimizer.step()
```

autograd keeps track!

```python
## x in range [0, 1]
x = torch.rand(3,2,requires_grad=True)
loss = torch.sum(torch.abs(x))
loss.backward()
## gradient should be all one
x.grad
```

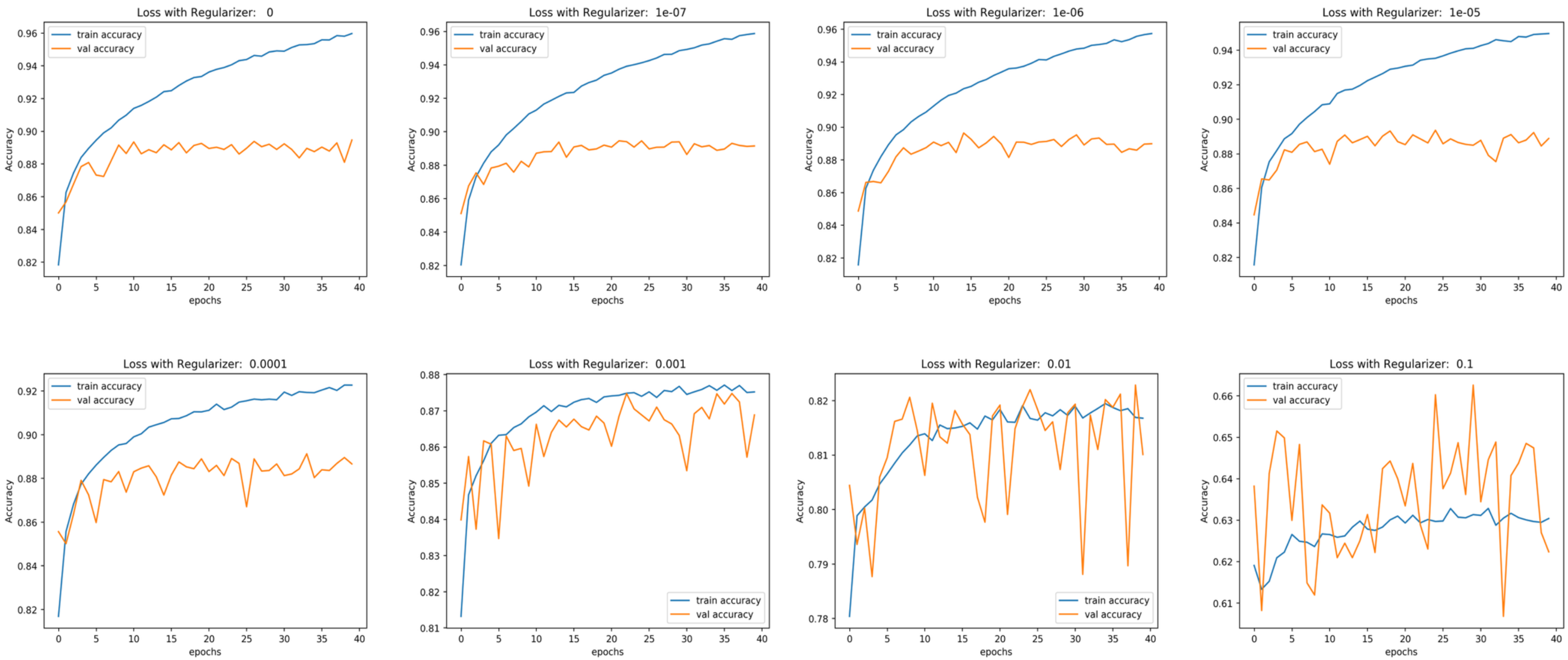
```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
```

LET'S TRY L2 REGULARIZATION…



just using regularization, we need λ~1e-3 to prevent over-fitting, but the loss is much higher (~0.45 vs 0.1)

LET'S TRY L2 REGULARIZATION…



same trend as the loss…
(note: this is with 80/20 train/loss split)

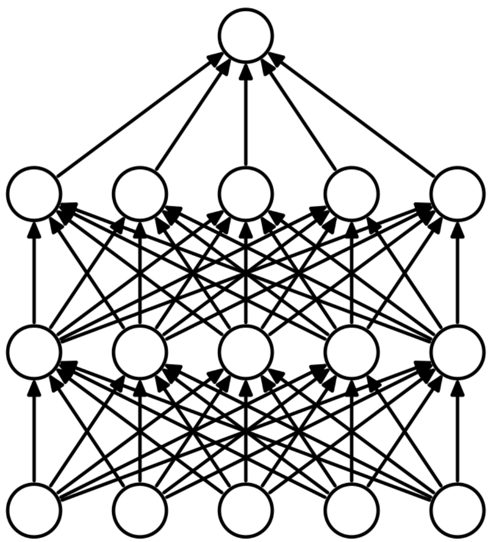
not totally satisfying!

DROPOUT REGULARIZATION

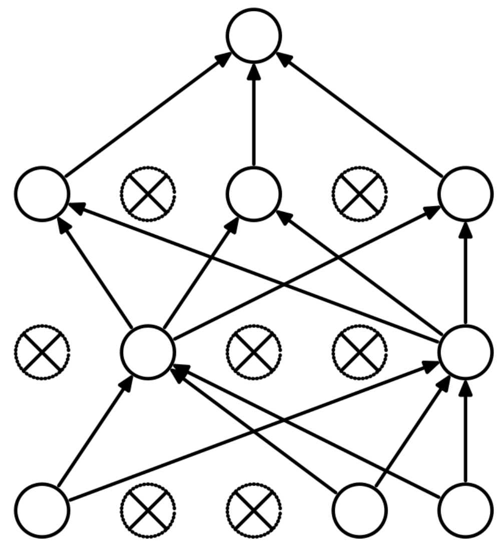
DROPOUT — A DIFFERENT TYPE OF REGULARIZATION

remove nodes in a layer with some dropout probability/rate

the **random pattern** is generated at the start of each mini-batch and held fixed during that mini-batch



(a) Standard Neural Net



(b) After applying dropout.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

DROPOUT

very effective at reducing over fitting and improving generalization

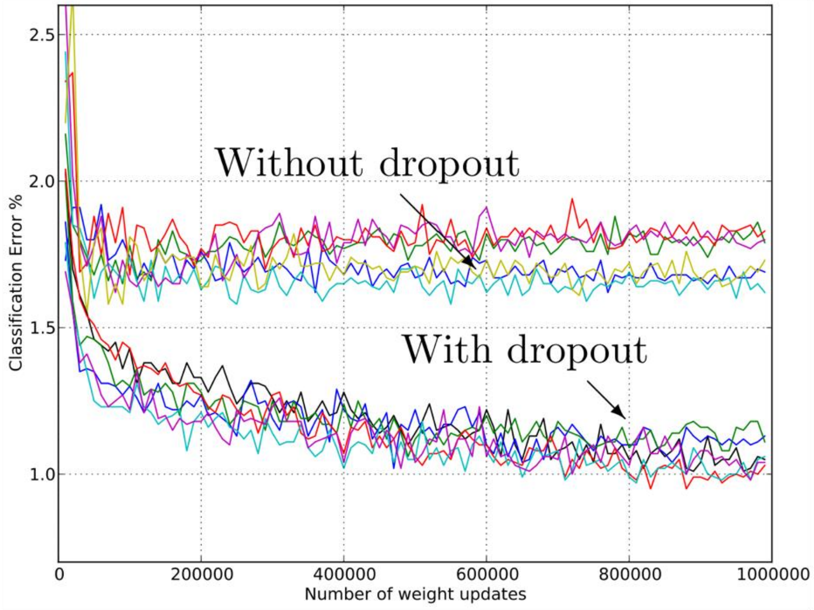
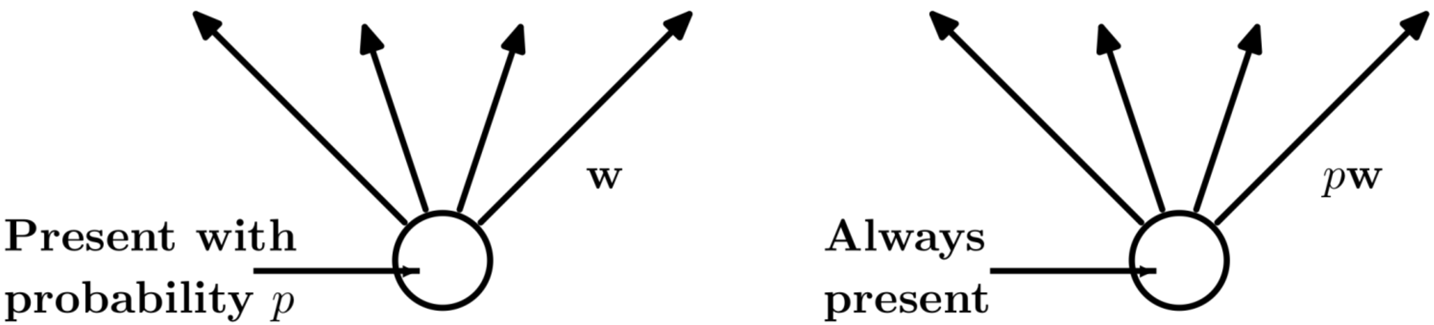


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

DROPOUT — ONLY DURING TRAINING!

Dropout is **used during training**, but in inference mode, all nodes are present



(a) At training time (b) At test time

for inference, replace the trained weights with $p \cdot w$, where $(1 - p)$ is the dropout rate

(ad hoc due to nonlinearities, but it works well enough)

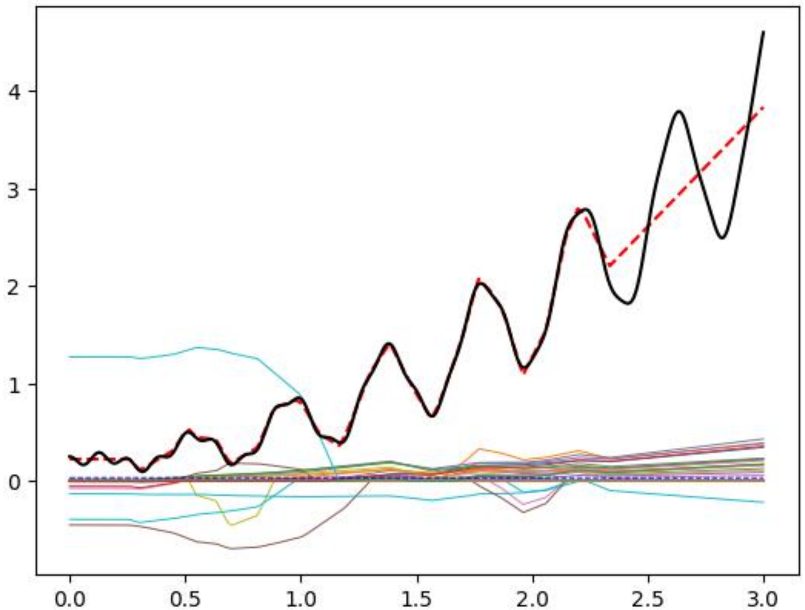
N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

DROPOUT EXAMPLE

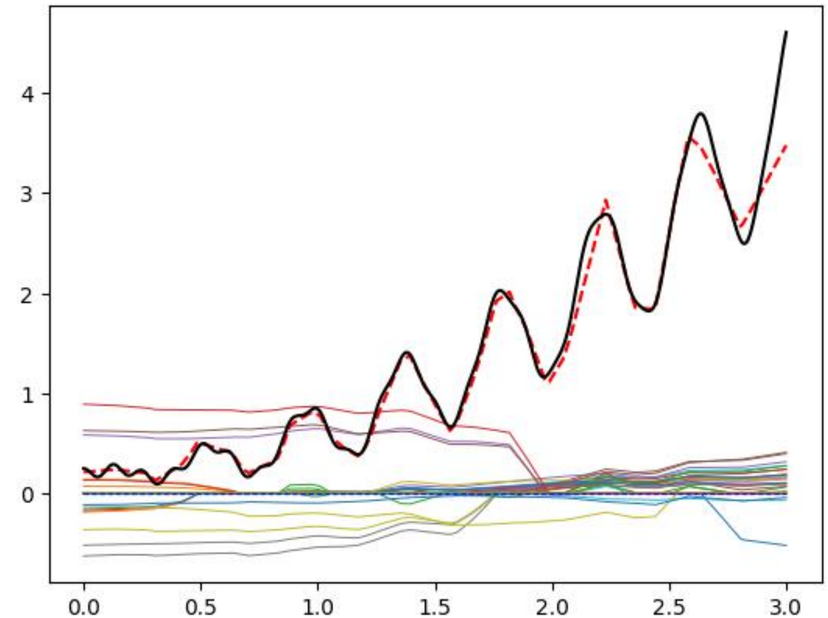
What happens when we train a neural net on Neilson's nonlinear function?

```
def neilson_example(x):
    return 0.2 + 0.4 * x**2 + 0.3 * x * np.sin(15 * x) + 0.05 * np.cos(50 * x)
```

3 hidden layers, 64 nodes each, ReLU activations



no dropout



20% Dropout

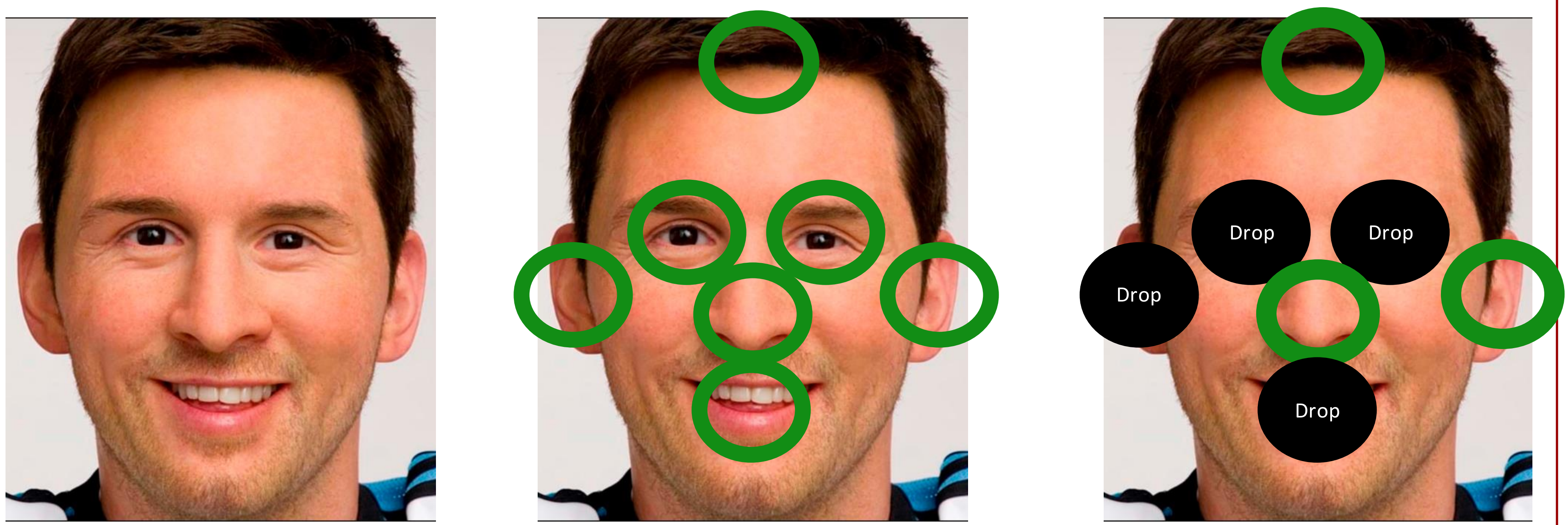
http://neuralnetworksanddeeplearning.com/chap4.html

DROPOUT INTUITION

Ensemble methods: train multiple networks for same task and average

Dropout can be viewed as an efficient way to do this in a single network

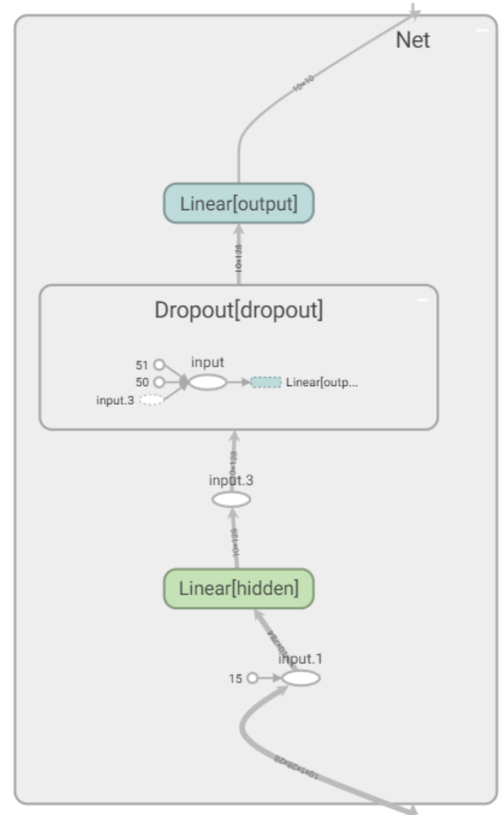
individual (or small groups of) nodes must do a reasonable job on he task w/o the deleted nodes lead to Robustness/Generalization



DROPOUT IN PYTORCH – JUST ANOTHER LAYER

```python
import torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden = nn.Linear(28*28, 128)
        self.dropout = nn.Dropout(p=0.3)
        self.output = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.hidden(x))
        x = self.dropout(x)
        x = self.output(x)
        return x

model = Net()

from torchsummary import summary
summary(model, input_size=(1, 1, 28*28))
```

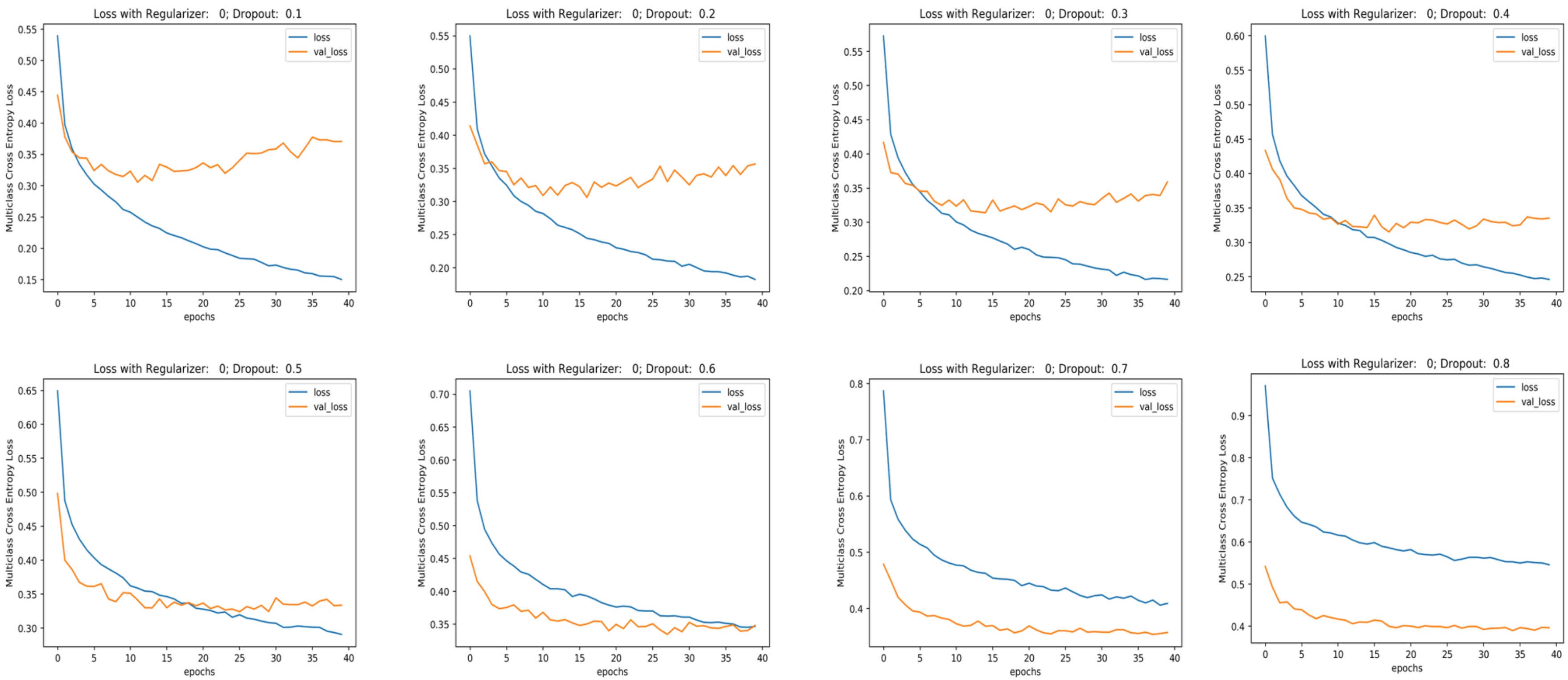
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1             [-1, 1, 1, 128]         100,480
           Dropout-2             [-1, 1, 1, 128]               0
            Linear-3              [-1, 1, 1, 10]           1,290
================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.39
Estimated Total Size (MB): 0.39
----------------------------------------------------------------
```



Dropout layer has <u>no trainable parameters</u> —it is an on/off *mask* that follows each node in the Dense layer

some layers have dropout built-in (*e.g.*, RNNs)

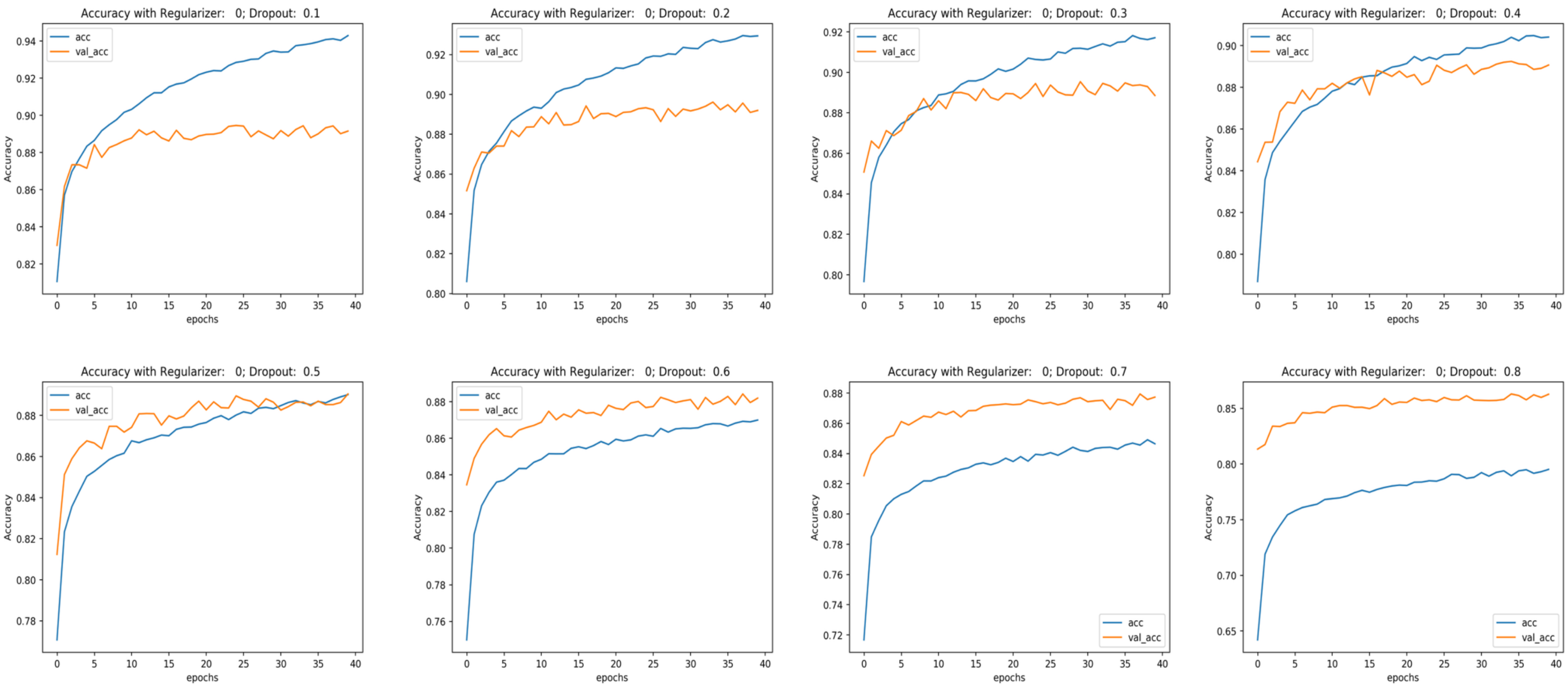
DROPOUT WITH NO L2 REGULARIZATION



with dropout of ~ 60%, we are not over-fitting and we have a loss of ~ 0.35

(better than L2 regularization in this case)

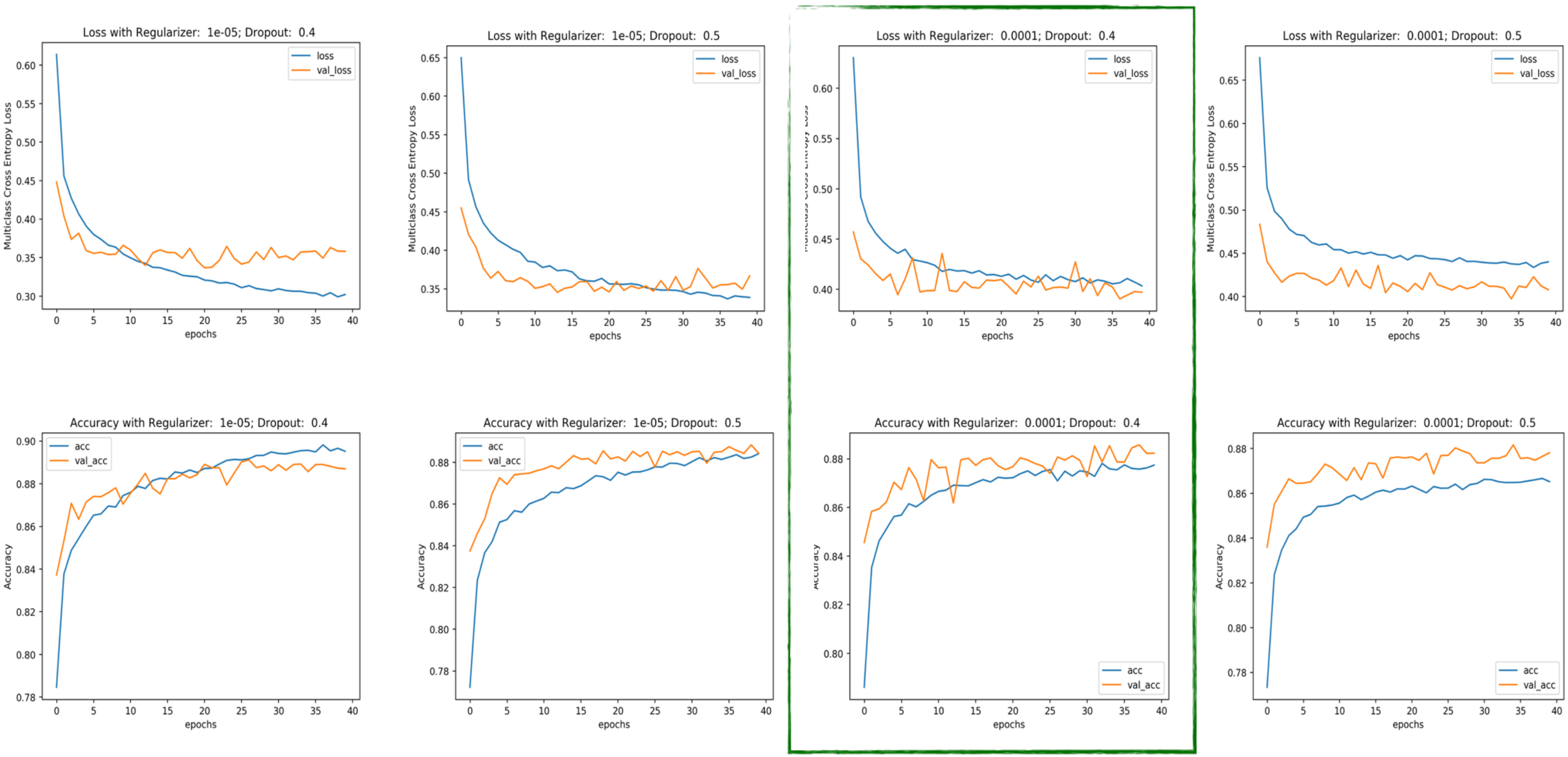
DROPOUT WITH NO L2 REGULARIZATION



similar trend as loss

(better than L2 regularization in this case)

DROPOUT AND L2 REGULARIZATION



best achieves test loss ~0.4, test accuracy ~ 88%

CONCLUSIONS FROM REGULARIZATION EXPERIMENTS

Main goal of Machine Learning is to
GENERALIZE

A combination of dropout and L2 regularization worked best

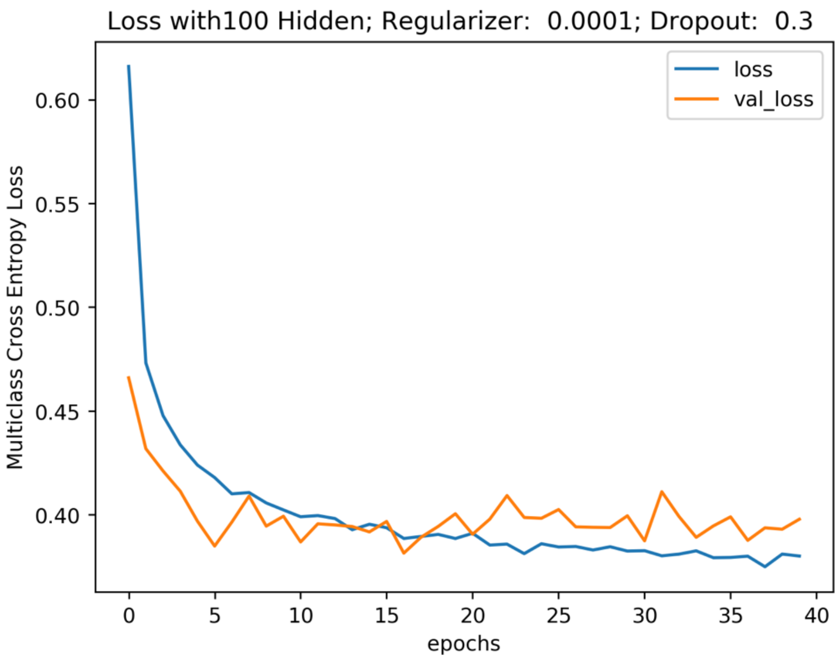
This required a pretty-high dropout rate plus regularization to not over-fit...

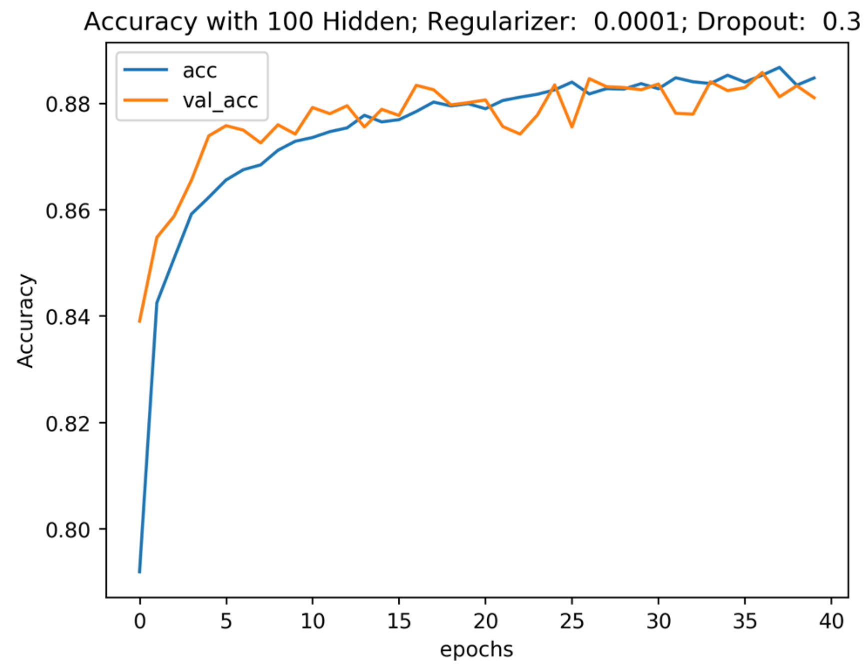
Note: we will see ~94% accuracy with CNNs on this problem

Nominal Values:
dropout rate: ≈20%
L2 Regularization: [1e-5, 1e-3]

What does this suggest to you??

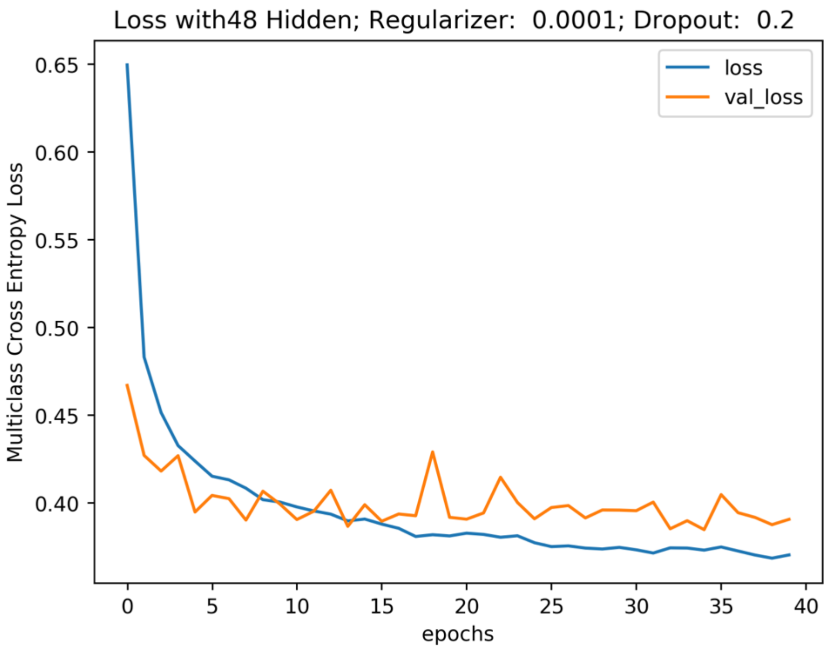
SMALLER MODEL, LESS REGULARIZATION

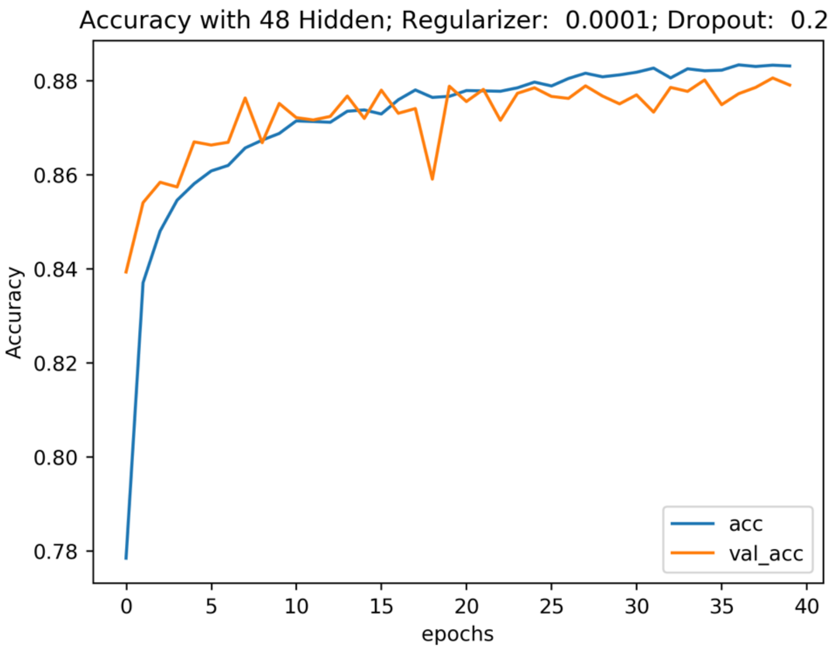




results with 100 hidden neurons

SMALLER MODEL, LESS REGULARIZATION



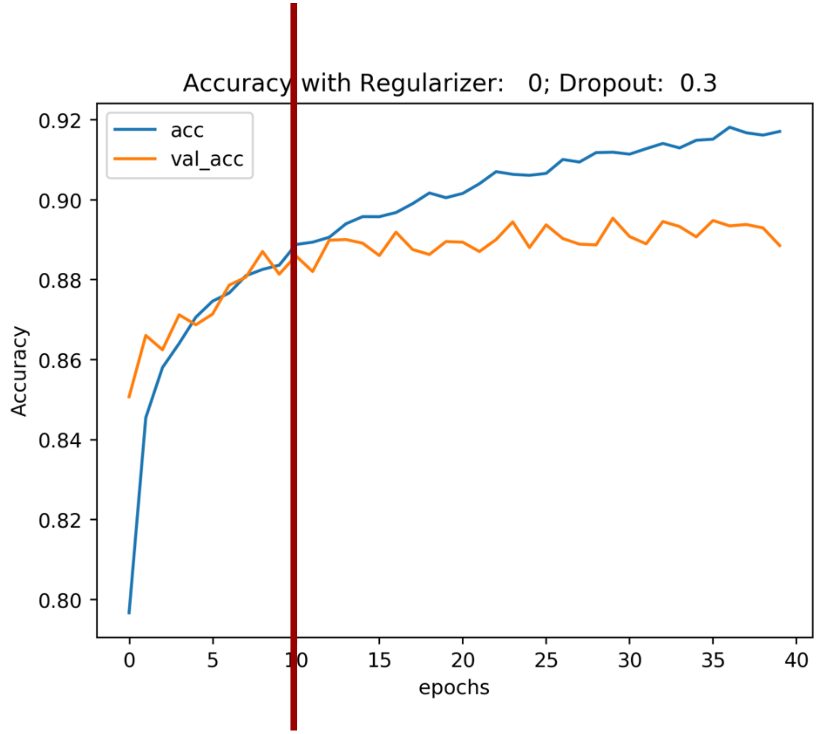


similar results with 48 hidden neurons

ANOTHER REGULARIZATION METHOD

“early stopping”

stop training when val starts performs *consistently* better than train



stop at ~10 epochs

OPTIMIZERS

OPTIMIZERS

Optimizers are **modifications** to standard Stochastic Gradient Descent (SGD)

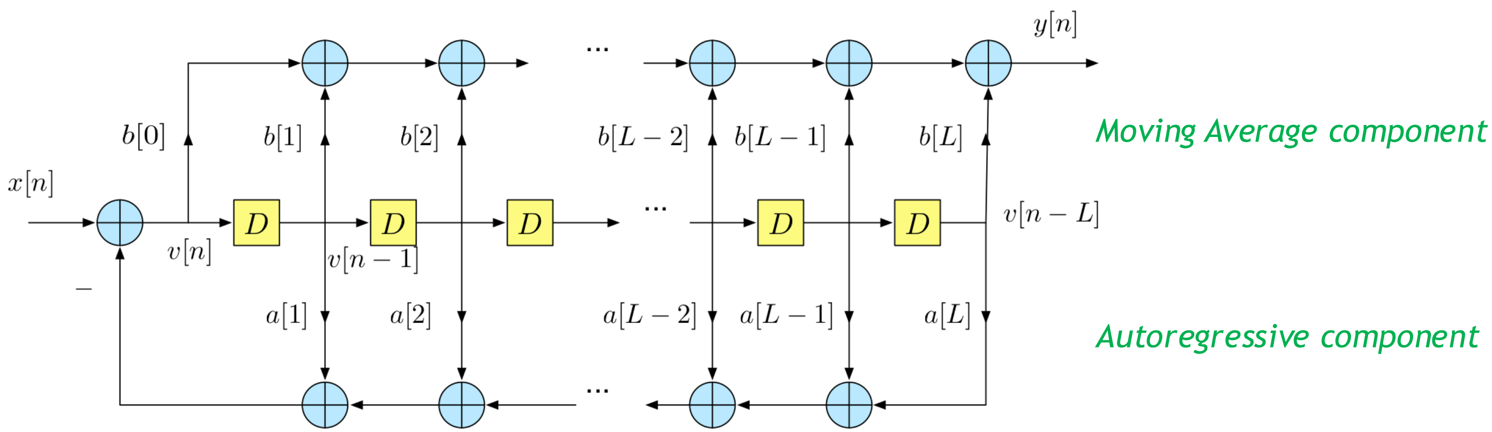
Three common modifications:

1. Gradient filtering
2. Gradient normalization
3. Learning rate schedule

- 1 and 2 usually associated with the “optimizer”
- learning rate schedule considered a separate parameter tuning task

LTI FILTER REVIEW

REVIEW OF ARMA LTI FILTERS



$$v[n] = x[n] - (a[1]v[n-1] + a[2]v[n-2] + \cdots a[L]v[n-L])$$

$$y[n] = b[0]v[n] + b[1]v[n-1] + b[2]v[n-2] + \cdots + b[L]v[n-L]$$

$$\text{state}[n] = (v[n-1], v[n-1], \ldots v[n-L])$$

implements this difference equation:

$$y[n] = \sum_{i=0}^{L} b[i]x[n-i] - \sum_{i=1}^{L} a[i]y[n-i]$$

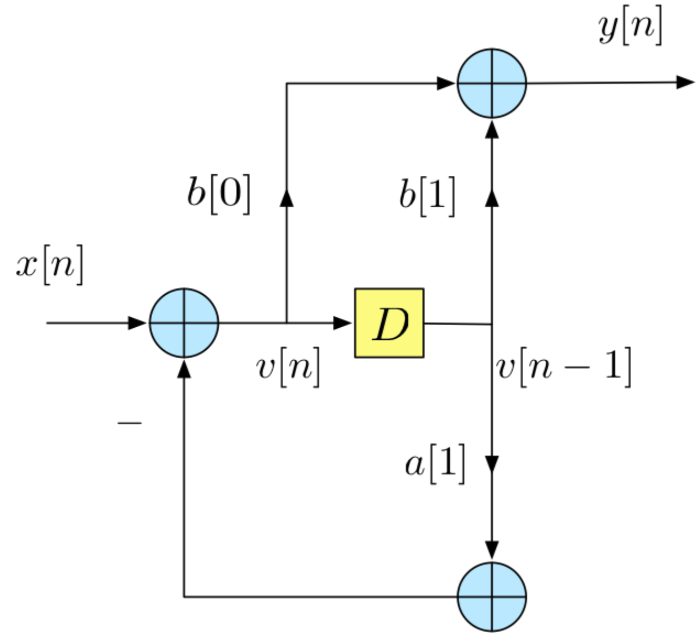
this is the canonical block diagram for an Lth order filter

Frequency response:

$$H(z) = \frac{b[0] + b[1]z^{-1} + b[2]z^{-2} \cdots + b[L]z^{-L}}{1 + a[1]z^{-1} + a[2]z^{-2} \cdots + a[L]z^{-L}} \qquad z = e^{j2\pi\nu}$$

REVIEW OF ARMA LTI FILTERS

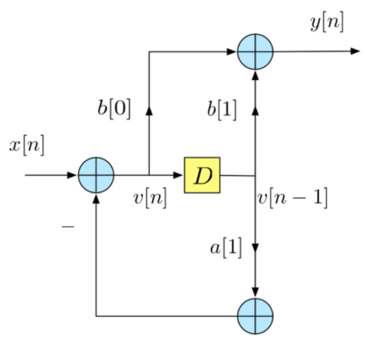
first order ARMA filter

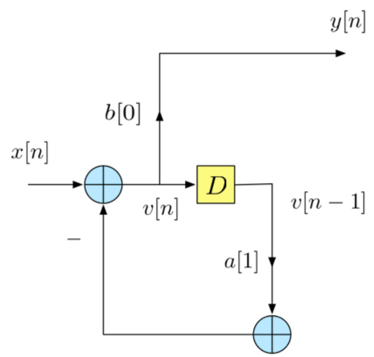


$$y[n] = -a[1]y[n-1] + b[0]x[n] + b[1]x[n-1]$$

$$H(z) = \frac{b[0] + b[1]z^{-1}}{1 + a[1]z^{-1}}$$

REVIEW OF FIRST ORDER LTI FILTERS





$$y[n] = -a[1]y[n-1] + b[0]x[n]$$

$$H(z) = \frac{b[0]}{1 + a[1]z^{-1}}$$

special cases for AR1:

Unit DC-Gain AR1: $y[n] = \alpha y[n-1] + (1-\alpha)x[n]$

$$H(z) = \frac{(1-\alpha)}{1 + \alpha z^{-1}}$$

has input-gain = $(1-\alpha)$

Unit input-Gain AR1: $y[n] = \alpha y[n-1] + x[n]$

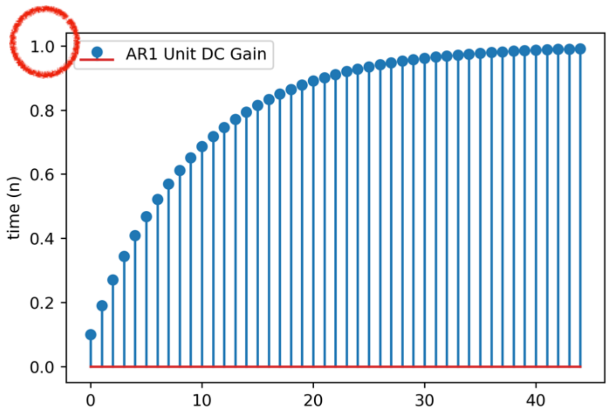
$$H(z) = \frac{1}{1 + \alpha z^{-1}}$$

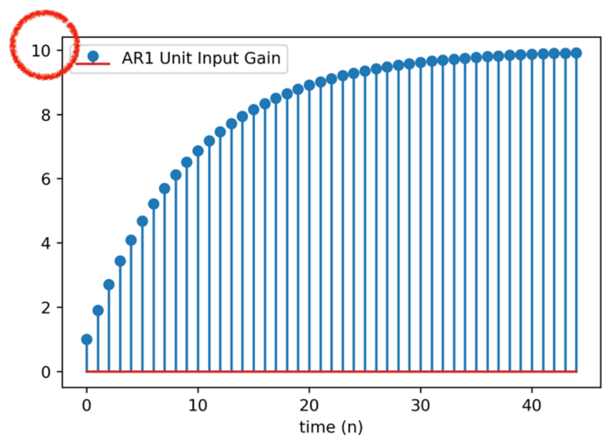
has DC-gain = $1/(1-\alpha)$

Recall: as α approaches 1, the filter gains memory and behaves as low-pass

REVIEW OF FIRST ORDER LTI FILTERS

unit step response with $\alpha = 0.9$





special cases for AR1:

Unit DC-Gain AR1:

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n]$$

$$s[n] = 1 - \alpha^{n+1}$$

$$H(z) = \frac{(1-\alpha)}{1+\alpha z^{-1}}$$

has input-gain = $(1-\alpha)$

Unit input-Gain AR1:

$$y[n] = \alpha y[n-1] + x[n]$$

$$s[n] = \frac{1-\alpha^{n+1}}{1-\alpha}$$

$$H(z) = \frac{1}{1+\alpha z^{-1}}$$

has DC-gain = $1/(1-\alpha)$

Recall: as α approaches 1, the filter gains memory and behaves as low-pass

TRANSIENT COMPENSATION

Unit DC Gain AR1: transient to reach steady state DC response

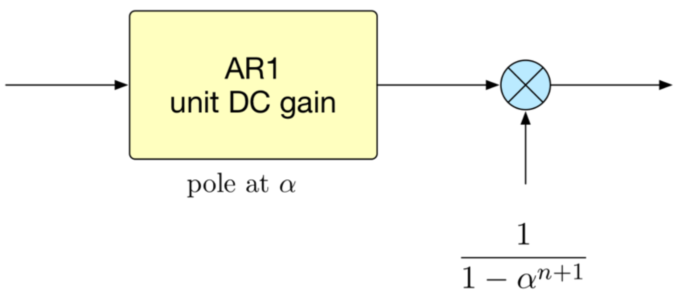
Unit input Gain AR1: pole dependent DC gain

$$y[n] = \alpha y[n-1] + (1-\alpha)x[n]$$

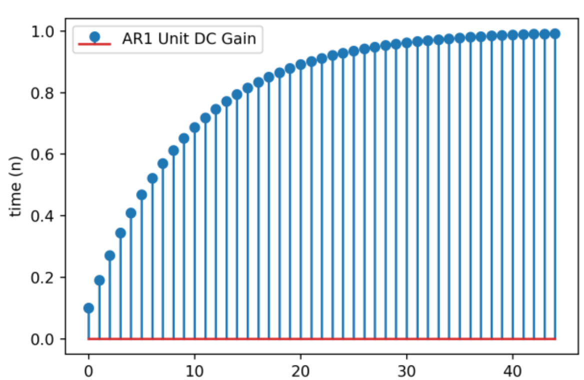
$$H(z) = \frac{(1-\alpha)}{1-\alpha z^{-1}}$$

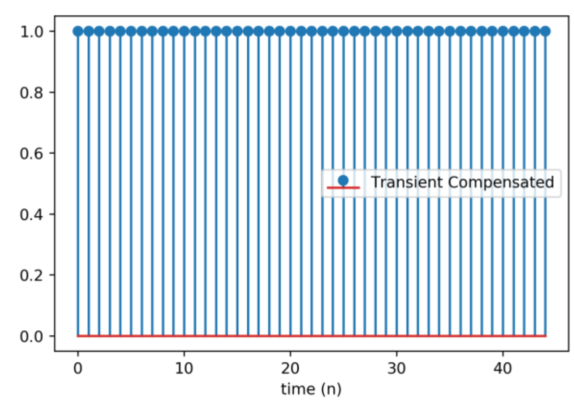
$$s[n] = 1 - \alpha^{n+1}$$

transient compensation



transient compensated step response



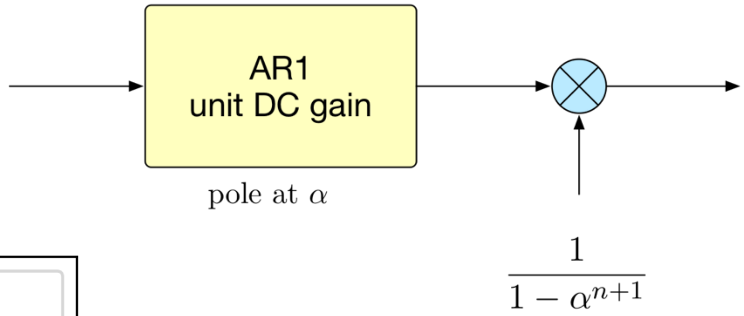


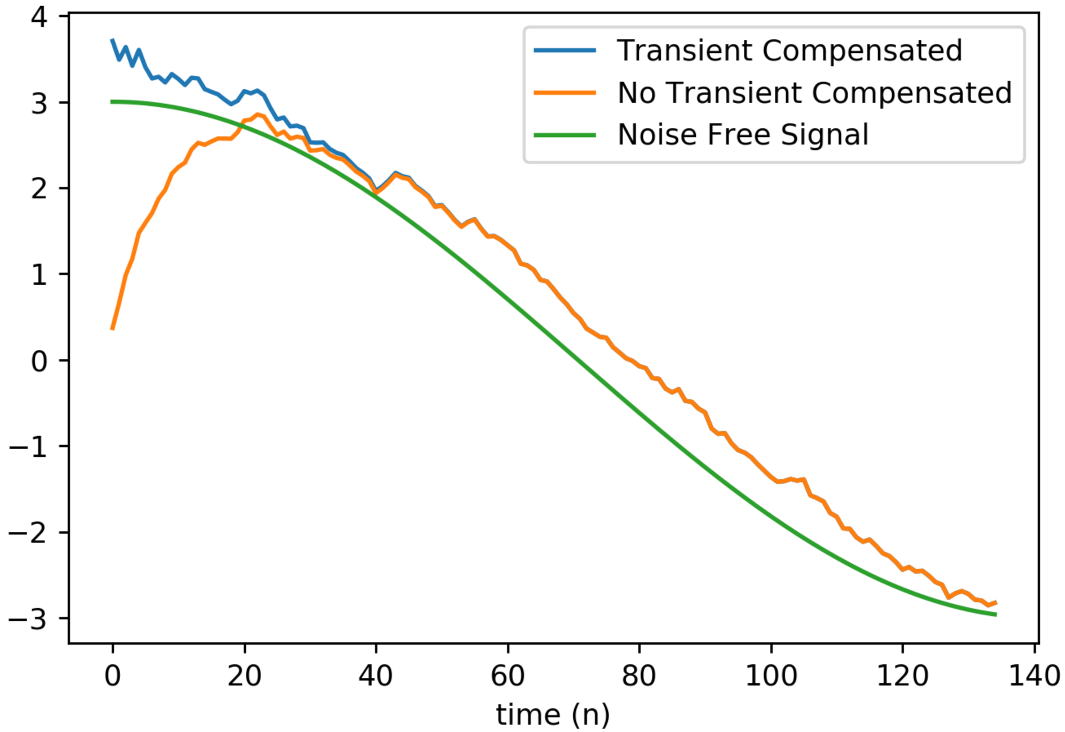
works for any scaled step input!

USC Viterbi School of Engineering

TRANSIENT COMPENSATION - NOISY EXAMPLE

Transient compensation





this example is a cosine in noise ($\alpha = 0.9$)

nice signal processing idea
(comes from deep learning AFAIK)

DEEP-LEARNING OPTIMIZERS

SUMMARY OF OPTIMIZERS

	gradient filtering	gradient normalization	grad variance filter	learning rate schedule
SGD	none	none	n/a	separate
SGD w/ momentum	AR1, unit input gain	none	n/a	separate
SGD w/ Nesterov Momentum	ARMA1 (1 pole, 1 zero)	none	n/a	separate
Adagrad	none	yes	summer	separate, but gradient norm does alter
Adadelta	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
RMSprop	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
Adam	AR1, unit input gain, transient compensation	yes	AR1, unit input gain, transient compensation	separate, but gradient norm does alter
Nadam (Adam w/ Nesterov)	ARMA1, transient compensation	yes	ARMA1, transient compensation	separate, but gradient norm does alter

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

GRADIENT FILTERING

GENERAL OPTIMIZER STRUCTURE + SGD

parameter update: $$\theta[i] = \theta[i-1] + \Delta[i]$$

i ~ indexes parameter updates ($i.e.$, mini-batch)

input step/gradient (update): $$\nabla[i] = \frac{\partial C}{\partial \theta[i-1]} \qquad g[i] = -\eta \frac{\partial C}{\partial \theta[i-1]}$$

SGD:

$$\Delta[i] = g[i]$$

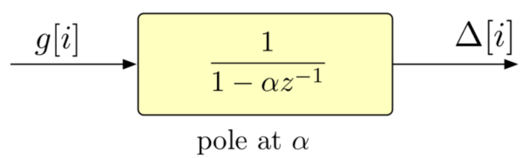
SGD with momentum:

$$v[i] = \alpha v[i-1] + g[i]$$

$$\Delta[i] = v[i]$$

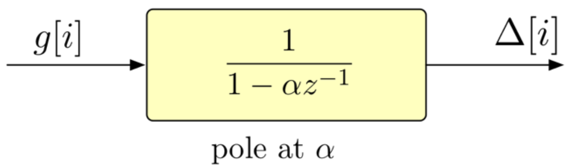
v is called the "*velocity*"

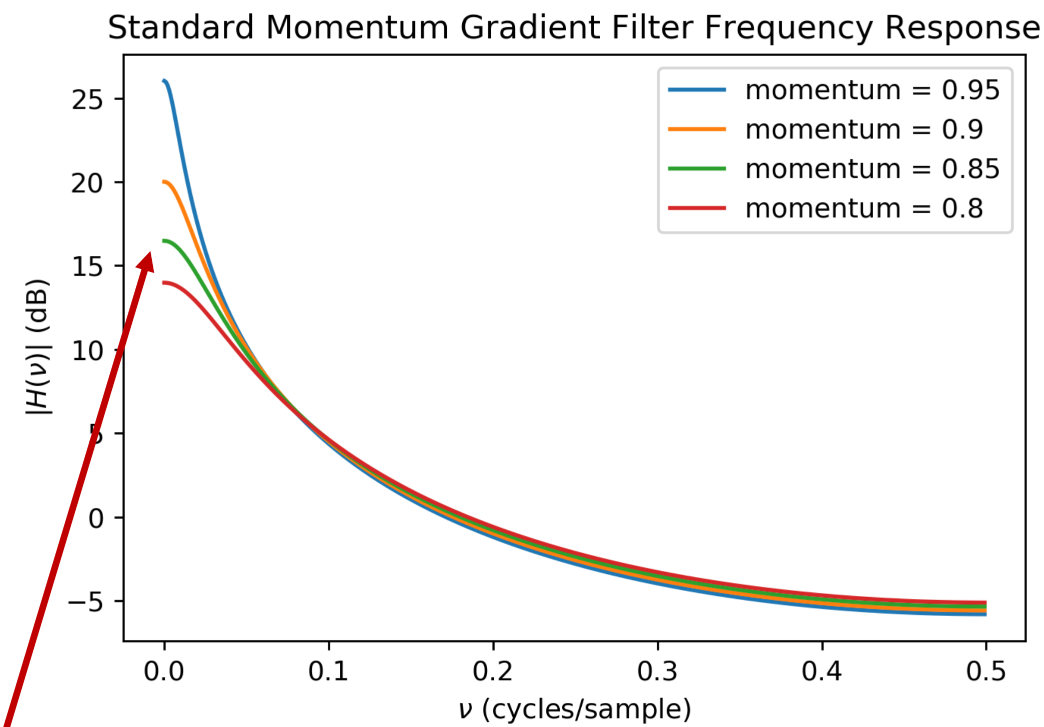
α is called "*momentum*" $(\alpha \sim 0.9)$



Momentum: low-pass filter on gradient —
removes high-frequency gradient noise

"STANDARD" MOMENTUM





Momentum: low-pass filter on the gradient removes high-frequency gradient noise

note that your momentum and learning rate are coupled

choosing larger momentum effectively increases your learning rate

SGD WITH NESTEROV MOMENTUM

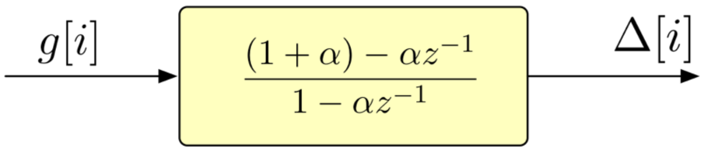
parameter update: $\theta[i] = \theta[i-1] + \Delta[i]$

input step/gradient (update): $\nabla[i] = \dfrac{\partial C}{\partial \theta[i-1]}$ $\qquad g[i] = -\eta \dfrac{\partial C}{\partial \theta[i-1]}$

$v[i] = \alpha v[i-1] + g[i]$ $\qquad$ v is called the "*velocity*"

$\Delta[i] = (1+\alpha)v[i] - \alpha v[i-1]$ $\qquad$ α is called "*momentum*"

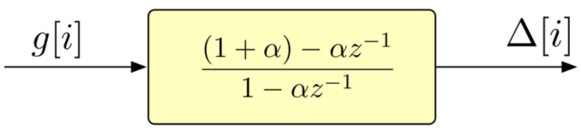
$\mathbf{(\alpha \sim 0.9)}$



pole at α

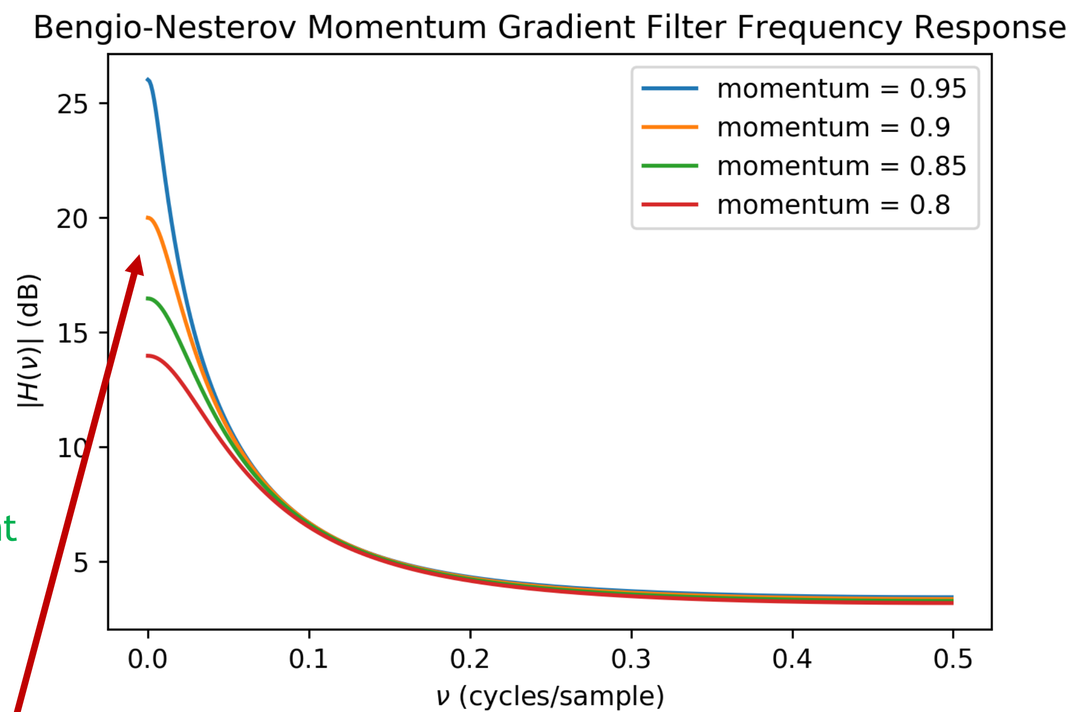
zero at $(1+\alpha)/\alpha$

SGD WITH NESTEROV MOMENTUM



pole at α

zero at $(1+\alpha)/\alpha$



Momentum: low-pass filter on the gradient removes high-frequency gradient noise

note that your momentum and learning rate are coupled

choosing larger momentum effectively increases your learning rate

STANDARD MOMENTUM VS NESTEROV MOMENTUM

standard

$$g[i] \longrightarrow \boxed{\frac{1}{1-\alpha z^{-1}}} \longrightarrow \Delta[i]$$

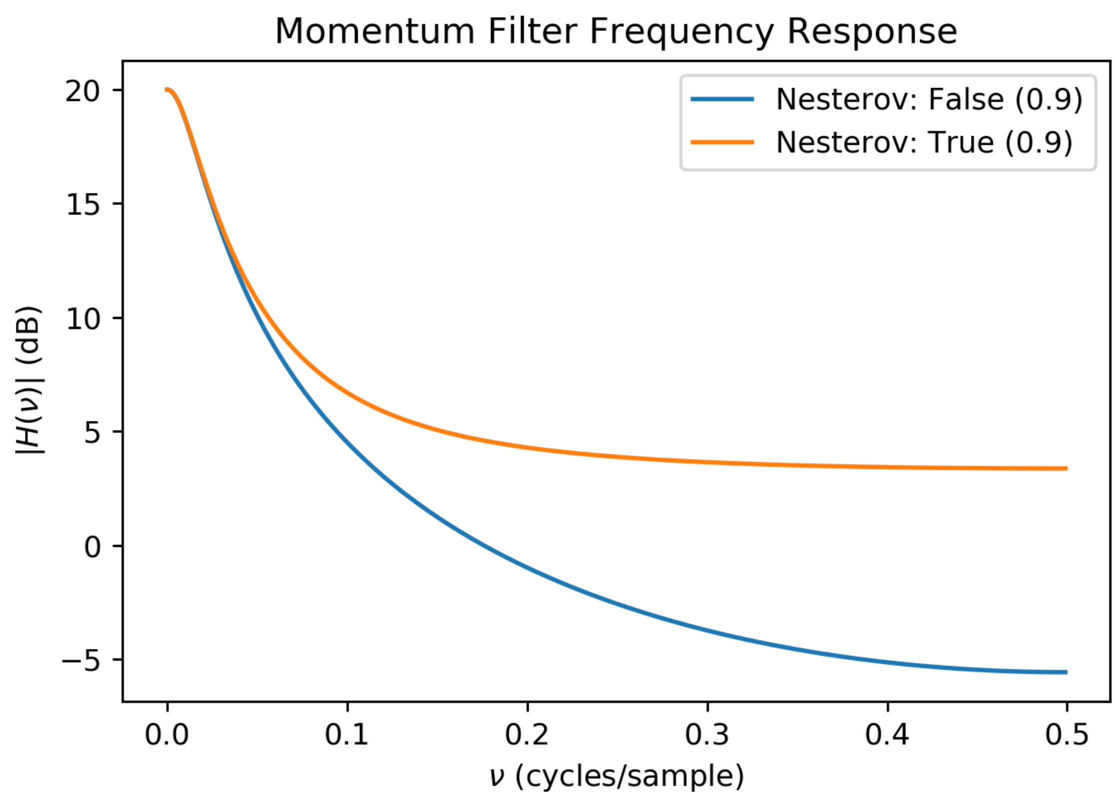
pole at α

Nesterov

$$g[i] \longrightarrow \boxed{\frac{(1+\alpha)-\alpha z^{-1}}{1-\alpha z^{-1}}} \longrightarrow \Delta[i]$$

pole at α

zero at $(1+\alpha)/\alpha$



standard momentum attenuates high frequencies more than Nesterov momentum

NESTEROV MOMENTUM (TYPICAL MOTIVATION)

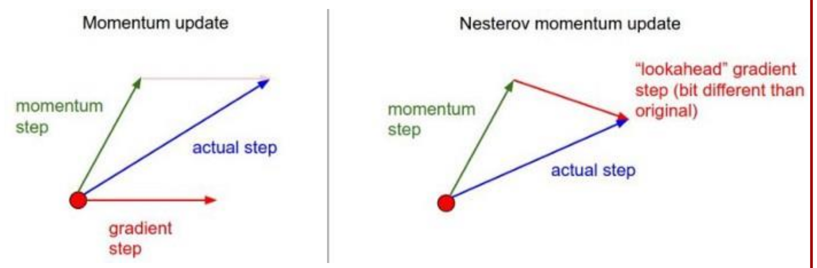
Motivated as compute "preliminary" parameter update before updating velocity and then adjust for velocity update

$$v_t = \mu_{t-1} v_{t-1} - \boxed{\epsilon_{t-1} \nabla f(\theta_{t-1} + \mu_{t-1} v_{t-1})}$$

$$\theta_t = \theta_{t-1} + v_t$$

what exactly is this??
… it's the *post-update* value

typical explanation



$$v_t = \mu_{t-1} v_{t-1} - \epsilon_{t-1} \nabla f(\boldsymbol{\Theta}_{t-1})$$

$$\boldsymbol{\Theta}_t = \boldsymbol{\Theta}_{t-1} - \mu_{t-1} v_{t-1} + \mu_t v_t + v_t$$

$$= \boldsymbol{\Theta}_{t-1} + \mu_t \mu_{t-1} v_{t-1} + (1 + \mu_t) \epsilon_{t-1} \nabla f(\boldsymbol{\Theta}_{t-1})$$

"Bengio's Formulation"

This is what PyTorch does!

Effect: adjust momentum coefficient invariant to learning rate

Bengio, Yoshua, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in optimizing recurrent networks." 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 2013.

https://jlmelville.github.io/mize/nesterov.html

NESTEROV MOMENTUM

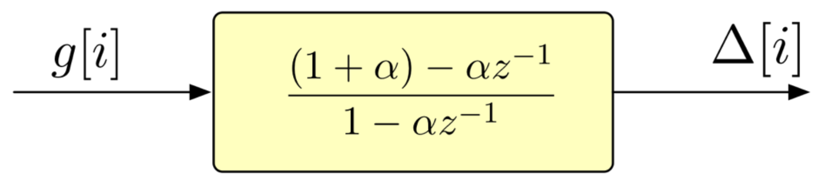
“Bengio’s Formulation”

$$v[i] = \alpha v[i-1] + g[i]$$

$$\theta[i] = \theta[i-1] + (1+\alpha)v[i] - \alpha v[i-1]$$

$$\Delta[i] = (1+\alpha)v[i] - \alpha v[i-1]$$

$$= v[i] + \alpha\, \underbrace{(v[i] - v[i-1])}_{\sim\text{acceleration}}$$



pole at α

zero at $(1+\alpha)/\alpha$

this formulation makes the pattern clear: choose any low-pass filter for this task — *i.e.*, optimize a second order ARMA filter (*e.g.*, Butterworth)

GRADIENT NORMALIZATION

GRADIENT NORMALIZATION

Idea: estimate gradient RMS and normalize

parameter update: $\theta[i] = \theta[i-1] + \Delta[i]$

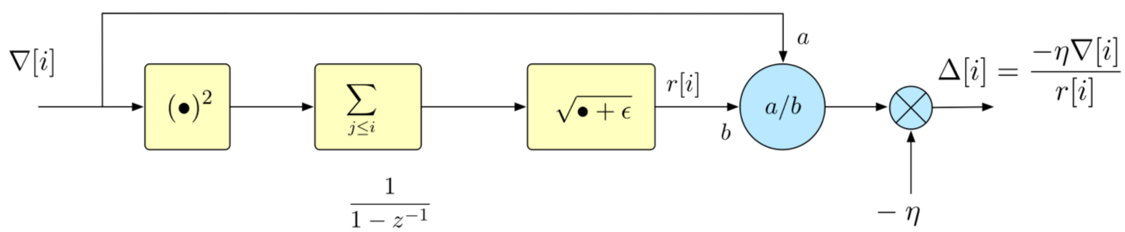
input step/gradient (update): $\nabla[i] = \dfrac{\partial C}{\partial \theta[i-1]}$ $\qquad g[i] = -\eta \dfrac{\partial C}{\partial \theta[i-1]}$

Can compute the RMS value of $\nabla[i]$ or $g[i]$

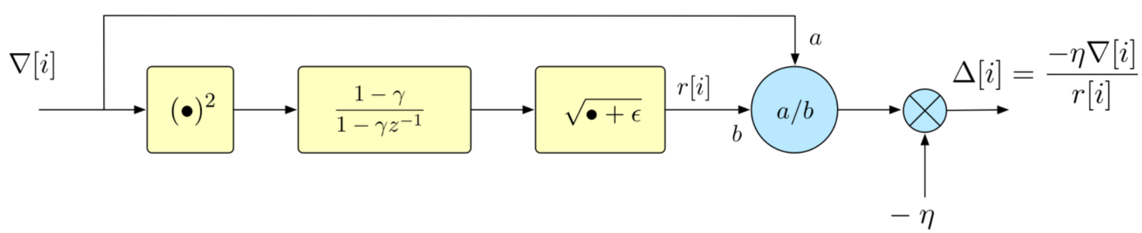
this is done by using a low-pass filter on the square of these quantities — *i.e.*, like computing the sample second moment

GRADIENT NORMALIZATION EXAMPLES

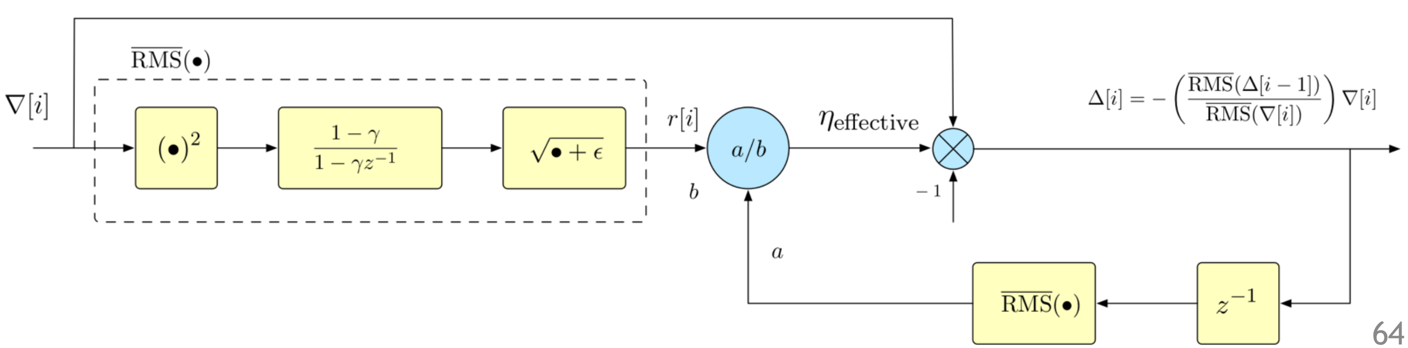
Adagrad:



RMSprop:



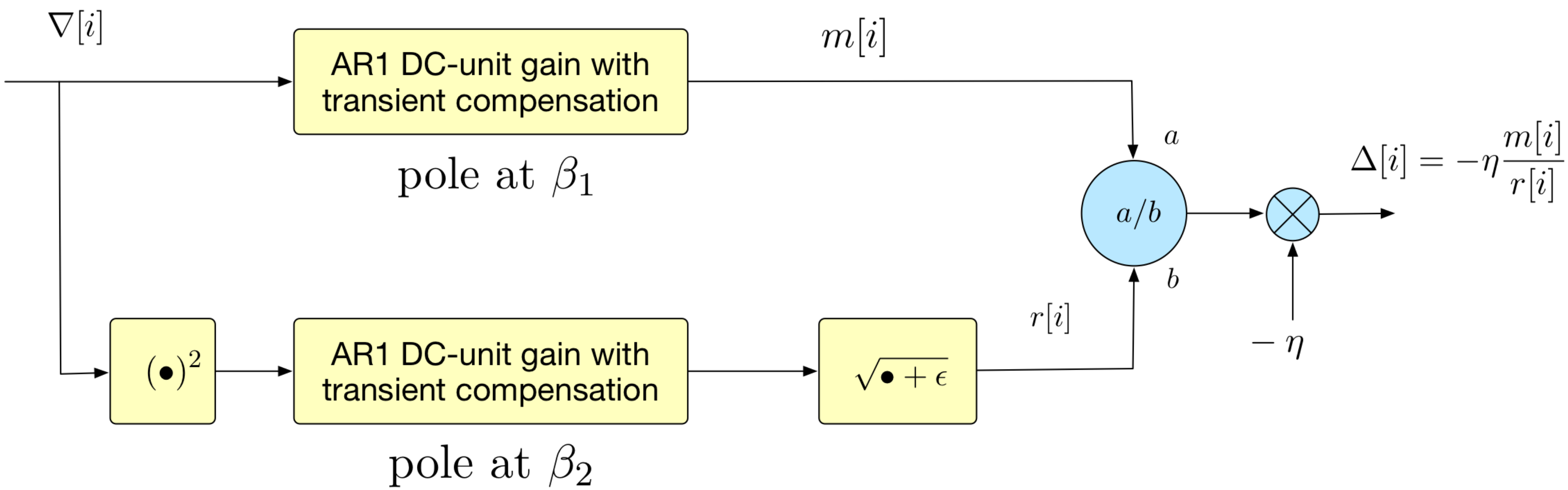
Adadelta:



ADAM (THE BEST OF ALL WORLDS?)

use unit-DC gain filters for gradient filtering **and** for computing the second moment

use transient compensation to reduce start-up effects of filters



D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

ADAM IMPLEMENTATION

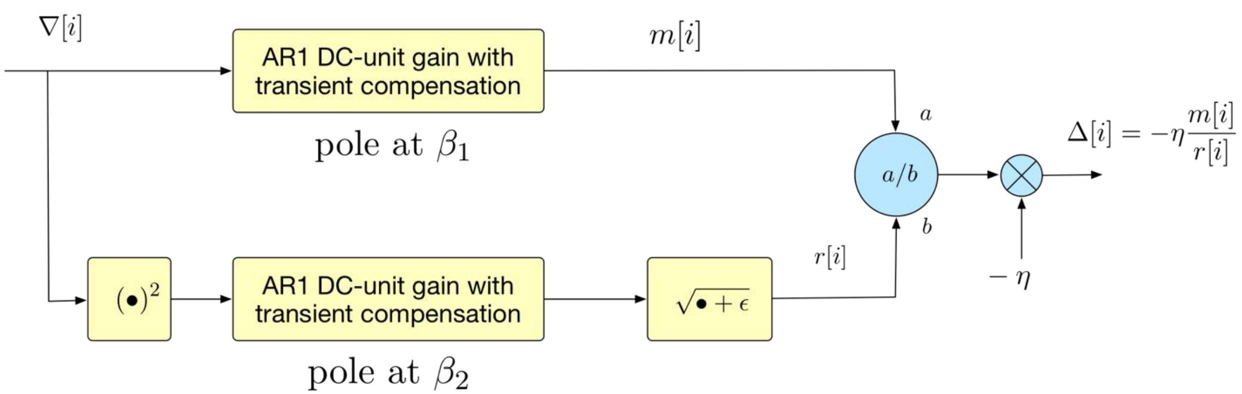
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

ADAM IN PYTORCH

https://pytorch.org/docs/stable/optim.html#torch.optim.Adam



Default:

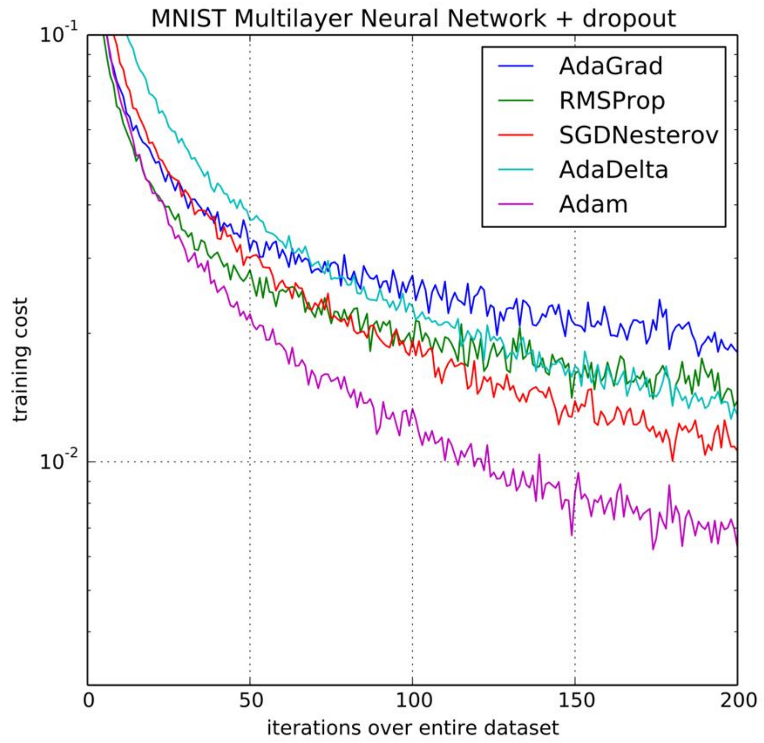
```
def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
             weight_decay=0, amsgrad=False):
```

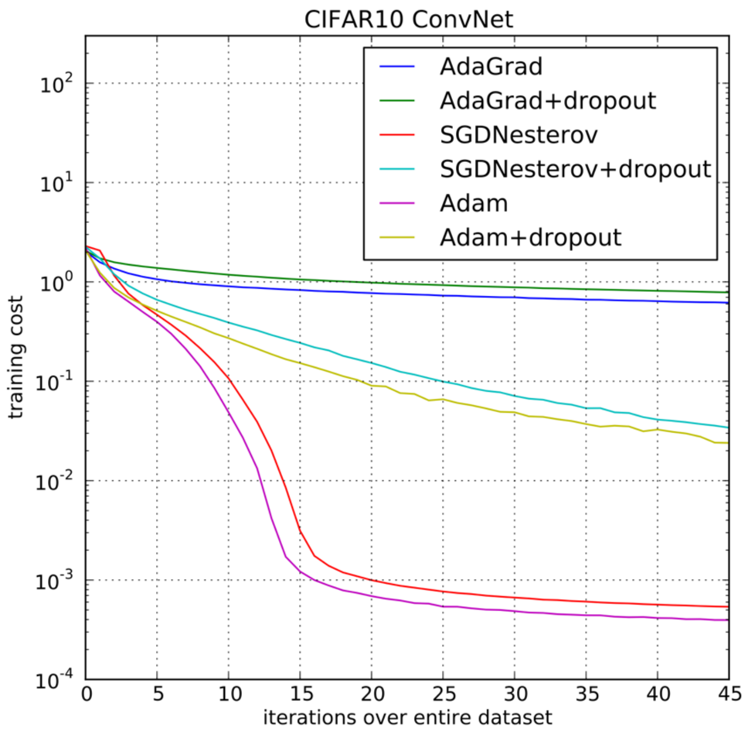
Tuned:

```
my_adam = optim.Adam(lr=0.002, betas=(0.92, 0.99),
eps=1e-09)
```

D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

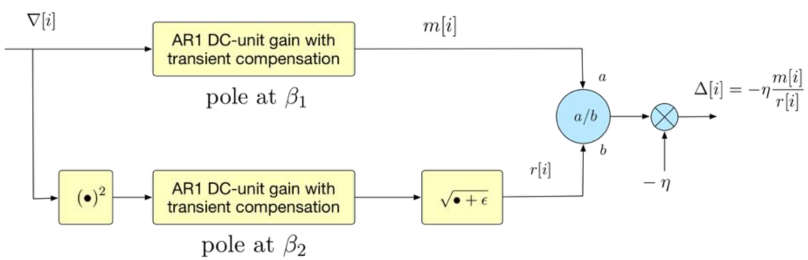
ADAM PERFORMANCE

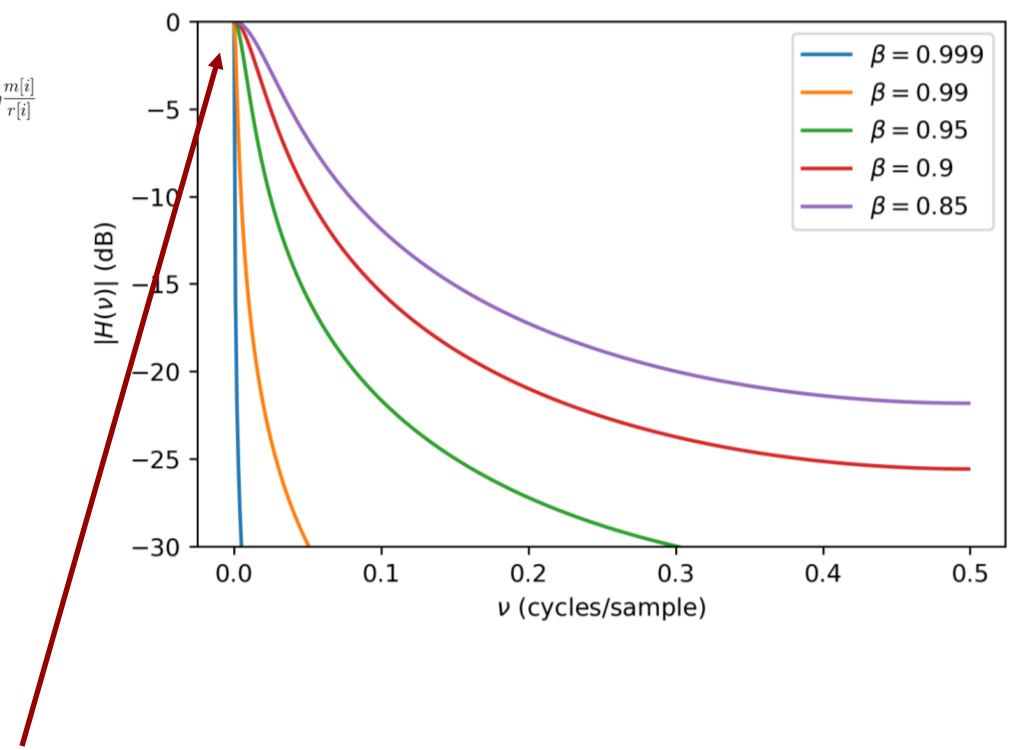


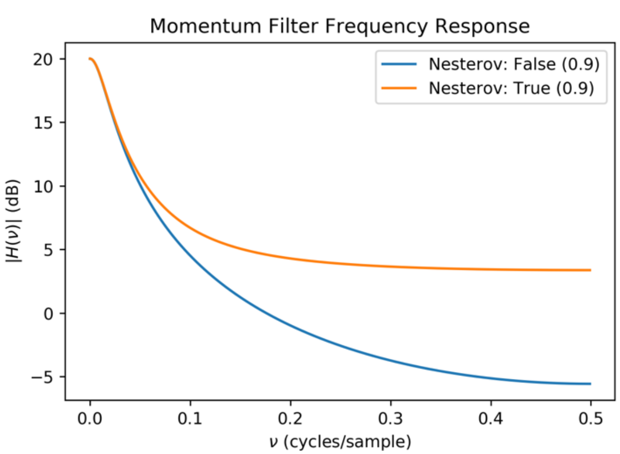


D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

ADAM GRADIENT FILTER FREQUENCY RESPONSE







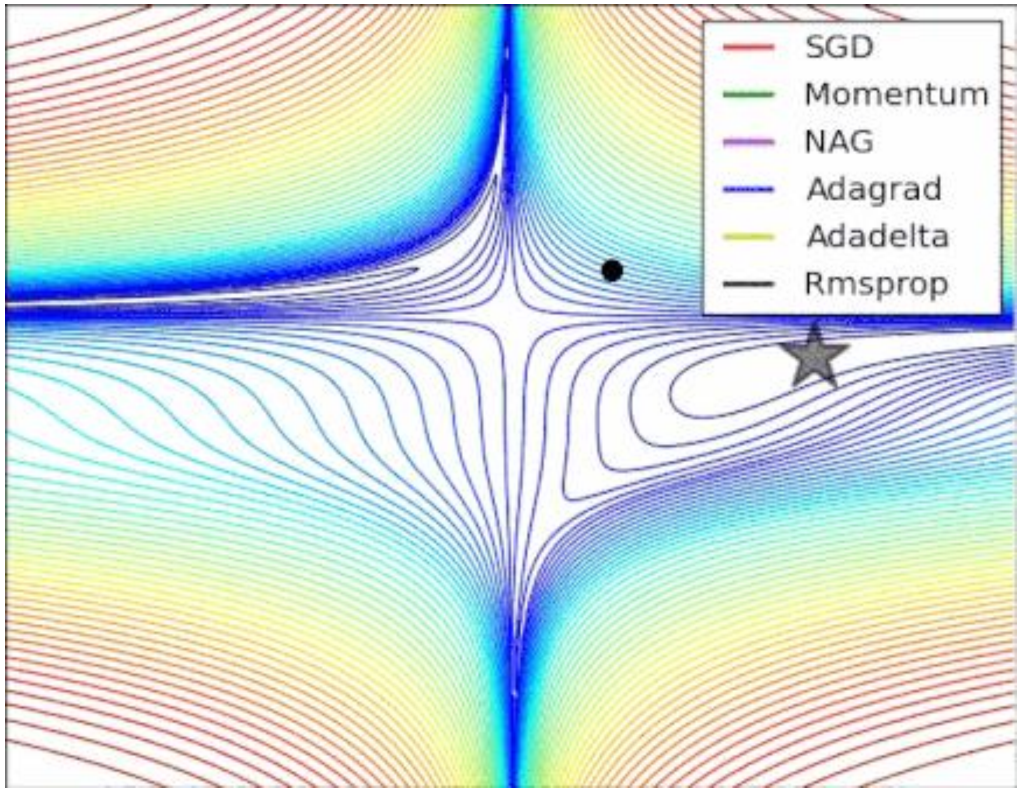
no coupling between momentum and learning rate!

SUMMARY OF OPTIMIZERS

	gradient filtering	gradient normalization	grad variance filter	learning rate schedule
SGD	none	none	n/a	separate
SGD w/ momentum	AR1, unit input gain	none	n/a	separate
SGD w/ Nesterov Momentum	ARMA1 (1 pole, 1 zero)	none	n/a	separate
Adagrad	none	yes	summer	separate, but gradient norm does alter
Adadelta	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
RMSprop	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
Adam	AR1, unit input gain, transient compensation	yes	AR1, unit input gain, transient compensation	separate, but gradient norm does alter
Nadam (Adam w/ Nesterov)	ARMA1, transient compensation	yes	ARMA1, transient compensation	separate, but gradient norm does alter

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

COMPARISON OF OPTIMIZERS



https://twitter.com/AlecRad

https://imgur.com/a/Hqolp

Visualization: https://vis.ensmallen.org/

USC Viterbi
School of Engineering

LEARNING RATE SCHEDULERS

LEARNING RATE SCHEDULES

Change (typically decrease) the learning rate as we do more parameter updates (batches)

Recall LMS: large learning rate implies faster convergences, but more “maladjustment error” (*i.e.*, *gradient noise*)

Could also use a LR schedule to try to force the optimizer out of a local minimum

(to go to a better local minimum, likely)

LEARNING RATE SCHEDULES IN PYTORCH

https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

```
learning_rate = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, nesterov=True)

# step_size: at how many multiples of epoch you decay
# step_size = 1, after every 1 epoch, new_lr = lr*gamma
# step_size = 2, after every 2 epoch, new_lr = lr*gamma

# gamma = decaying factor
scheduler = StepLR(optimizer, step_size=1, gamma=0.1)

for epoch in range(num_epochs):
    [...]
    # Decay Learning Rate
    scheduler.step()
    # Print Learning Rate
    print('Epoch:', epoch,'LR:', scheduler.get_lr())
```

Rule: apply learning rate **scheduling**
AFTER optimizer update

```
>>> scheduler = ...
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

From LMS, we know that large learning rate implies faster convergences, but more “maladjustment error” (*i.e.*, gradient noise)

USC Viterbi School of Engineering

COMMON LR SCHEDULES

$$\eta_i = \rho \eta_0$$

Exponential Decay

$$\eta_i = \eta_0 \left(1 - \frac{i}{N_{\text{epochs}}}\right)$$

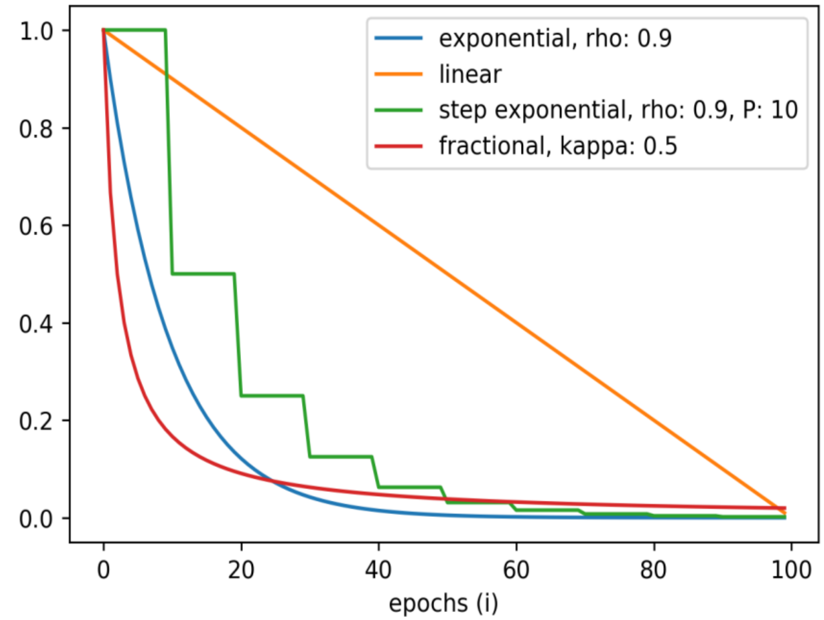
Linear Decay

$$\eta_i = \eta_0 \rho^{\lfloor i/P \rfloor}$$

Step Exponential Decay

$$\eta_i = \frac{\eta_0}{1 + \kappa i}$$

Fractional Decay



$$0 \le \rho \le 1 \qquad \kappa > 0$$

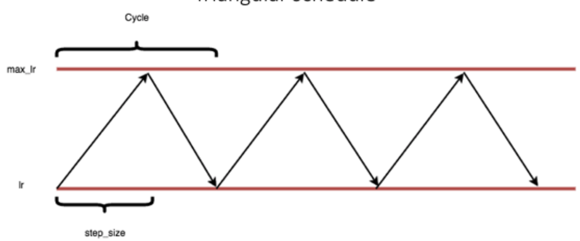
Another common LR schedule is to decrease the LR at specific epochs in a stepwise manner

$e.g.$, every 10 epochs: $\eta \leftarrow 0.1 \cdot \eta$

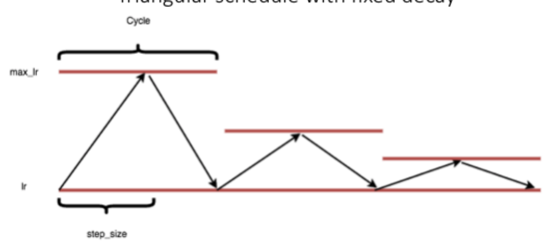
EXOTIC "ANNEALING" LR SCHEDULES

Triangular Schedules

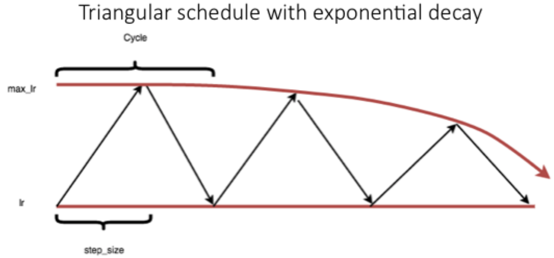
Triangular schedule



Triangular schedule with fixed decay



Triangular schedule with exponential decay



L. N. Smith, "Cyclical Learning Rates for Training Neural Networks", arXiv:1506.01186

Cosine Schedules

$$\eta_t = \eta^i_{\min} + \frac{1}{2}\left(\eta^i_{\min} - \eta^i_{\max}\right)\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right)$$

Loshchilov, Ilya, and Frank Hutter. "SGDR: Stochastic gradient descent with warm restarts." arXiv preprint arXiv:1608.03983 (2016).

cosine annealing schedule in PyTorch

https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.CosineAnnealingLR

cosine annealing with "warm restarts"

torch.optim.lr_scheduler.CosineAnnealingWarmRestarts

https://www.jeremyjordan.me/nn-learning-rate/

TOPIC OUTLINE

- Universal Approximation Theorem
    - Why Deep?
- A Gentle Introduction to PyTorch
- Vanishing gradient and activations
- Weight initialization
- Cost functions, regularization, dropout
- Optimizers
- Batch Normalization
- Hyperparameter optimization

BATCH NORMALIZATION

BATCH NORMALIZATION LAYER

learn the best “level" for internal activations

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

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

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

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

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

γ and β are trainable parameters

this normalization is done for each mini-batch but what to do when using trained network for inference?

During inference, replace the mini-batch data-average mean and variance by the data-average mean and variance over the entire dataset

11: In $N_{\mathrm{BN}}^{\mathrm{inf}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$

commonly used and effective technique in deep CNNs

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

TOPIC OUTLINE

- Universal Approximation Theorem
  - Why Deep?
- A Gentle Introduction to PyTorch
- Vanishing gradient and activations
- Weight initialization
- Cost functions, regularization, dropout
- Optimizers
- Batch Normalization
- Hyperparameter optimization

USC Viterbi
School of Engineering

HYPERPARAMTER OPTIMIZATION

THIS IS HOPELESSLY COMPLEX!?!?!

We need to search over:

1. Model Architecture
   1. Number of layers
   2. Layer types
   3. Number of nodes in each layer
2. Loss Functions
3. Regularization Methods
   1. L1, L2, L1_L2
   2. Vary with layer
   3. Weight vs bias
4. Optimizers
   1. Type: SGD, Adam, etc.
   2. Parameters
   3. Learning rate schedules

USC Viterbi School of Engineering

FOLLOW HIGH-LEVEL GUIDELINES

Loss Function

Binary Classification → Use sigmoid output activation with Binary Cross Entropy Loss

M-ary Classification → Use softmax output activation with Multi-Class Cross Entropy Loss

Regression → Use linear output activation with MSE loss (L2)

Regularization → Use dropout and L2 regularization

Target network size so that:

dropout rate ~ 0.2, L2-reg coefficient ~ 1e-4

Optimizer → Adam with defaults is a good start

`optim.lr_scheduler.ReduceLROnPlateau()`
or simple step LR schedules

A lot of focus on this in the literature, but designing your dataset is more important (consider above fine tuning in practice)

AUTOMATED NETWORK ARCHITECTURE SEARCH AND HYPERPARAMETER OPTIMIZATION

Approach combines Bayesian optimization with grid search while targeting a combination of classification accuracy and runtime complexity (CNNs)