Training Methods

EE599 Deep Learning

Keith M. Chugg
Spring 2020

USC University of Southern California

Outline for Slides

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

Universal Approximation Theorem

Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there exist an integer m_1 and sets of real constants α_i, b_i, and w_{ij}, where $i = 1, ..., m_1$ and $j = 1, ..., m_0$ such that we may define

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \varphi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i \right) \tag{4.88}$$

as an approximate realization of the function $f(\cdot)$; that is,

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \varepsilon$$

for all $x_1, x_2, \dots, x_{m_0}$ that lie in the input space.

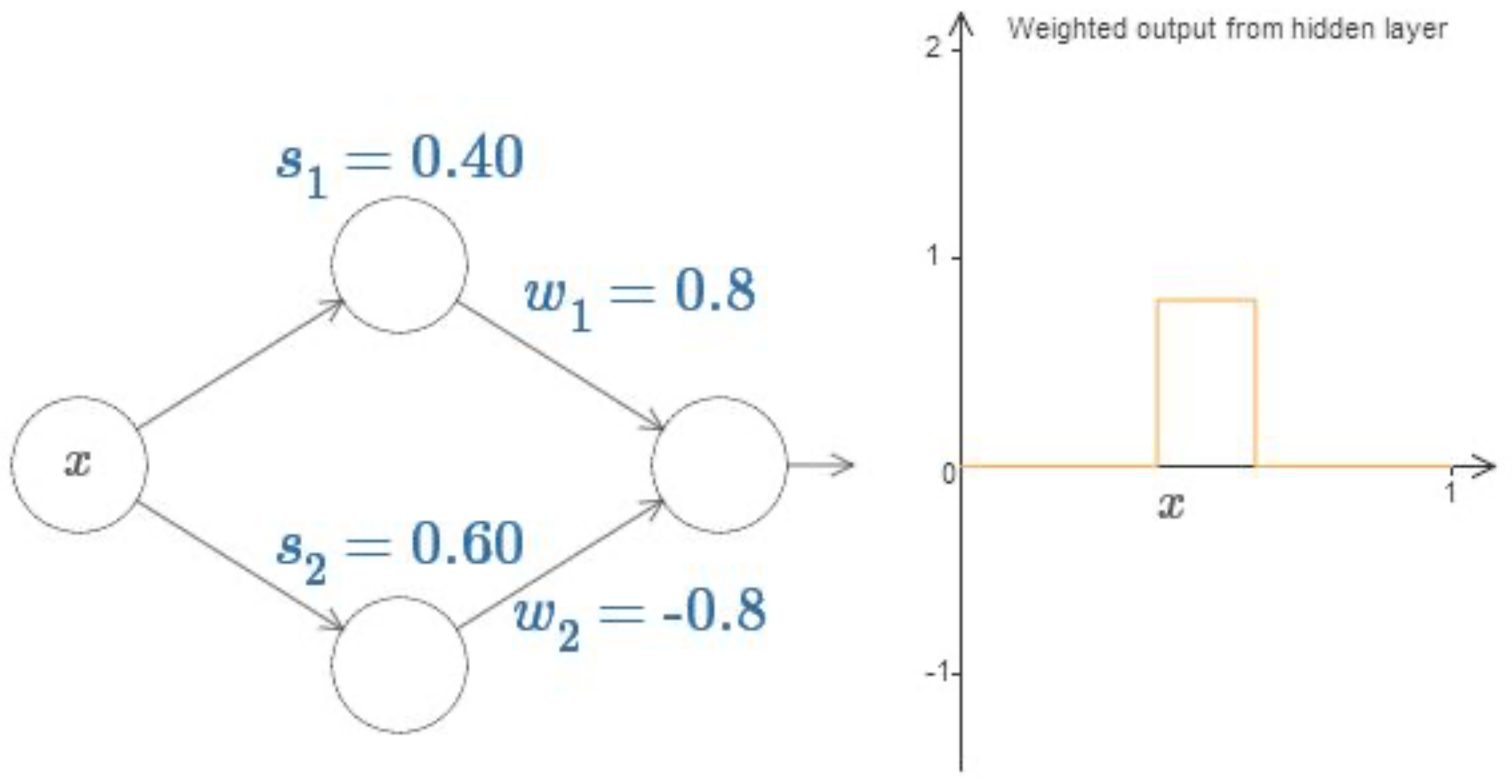
A single hidden layer MLP with squashing activation in the hidden layer and linear output layer can approximate any "engineering function"

[Haykin] Simon Haykin, Neural Networks And Learning Machines 3rd Edition, Pearson, 2009.

Universal Approximation Theorem

how does the intuition behind this work?

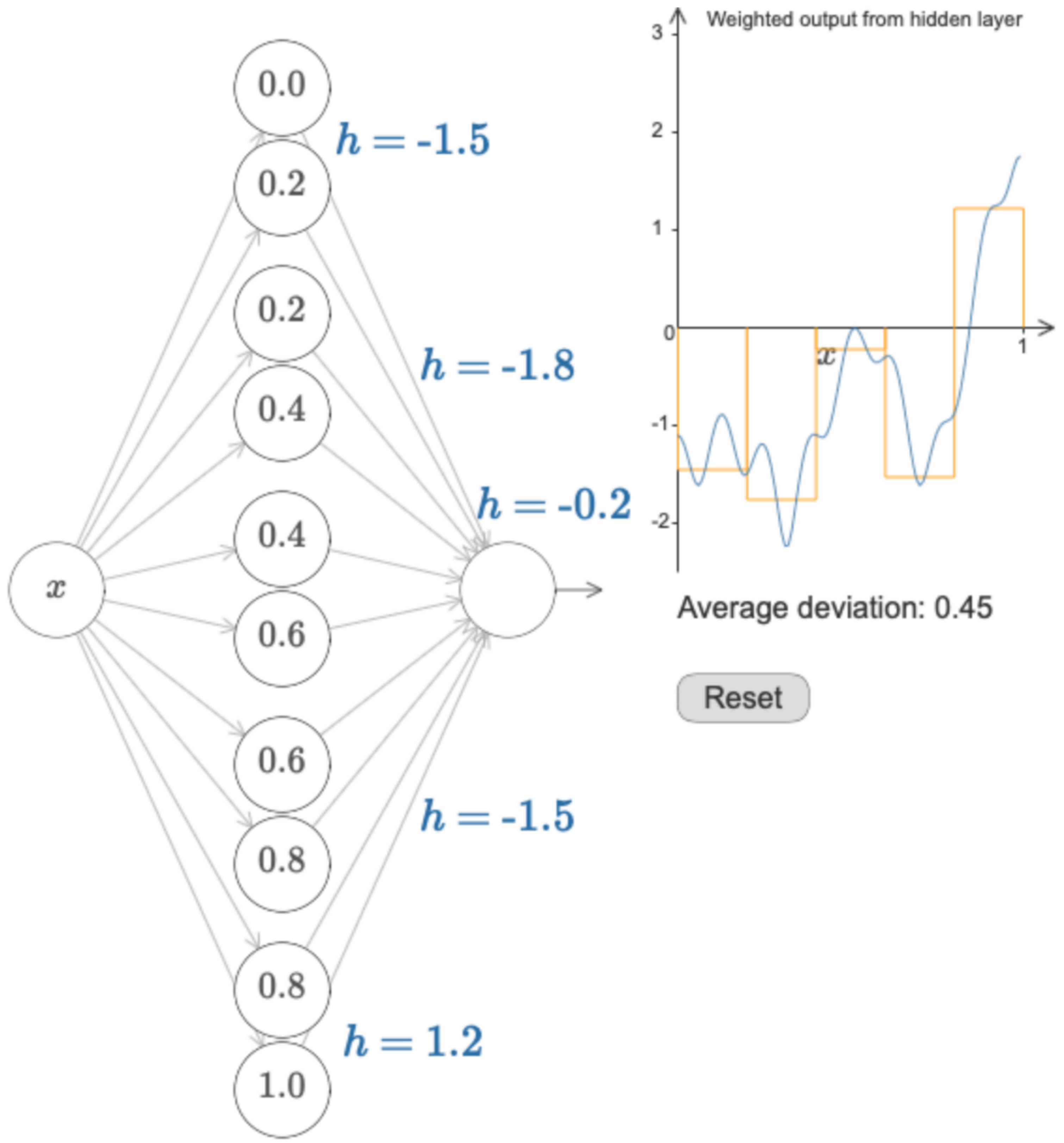
http://neuralnetworksanddeeplearning.com/chap4.html



can create a
"bump" function

done by choosing large
weights in layer 1

s = -b/w (step position)

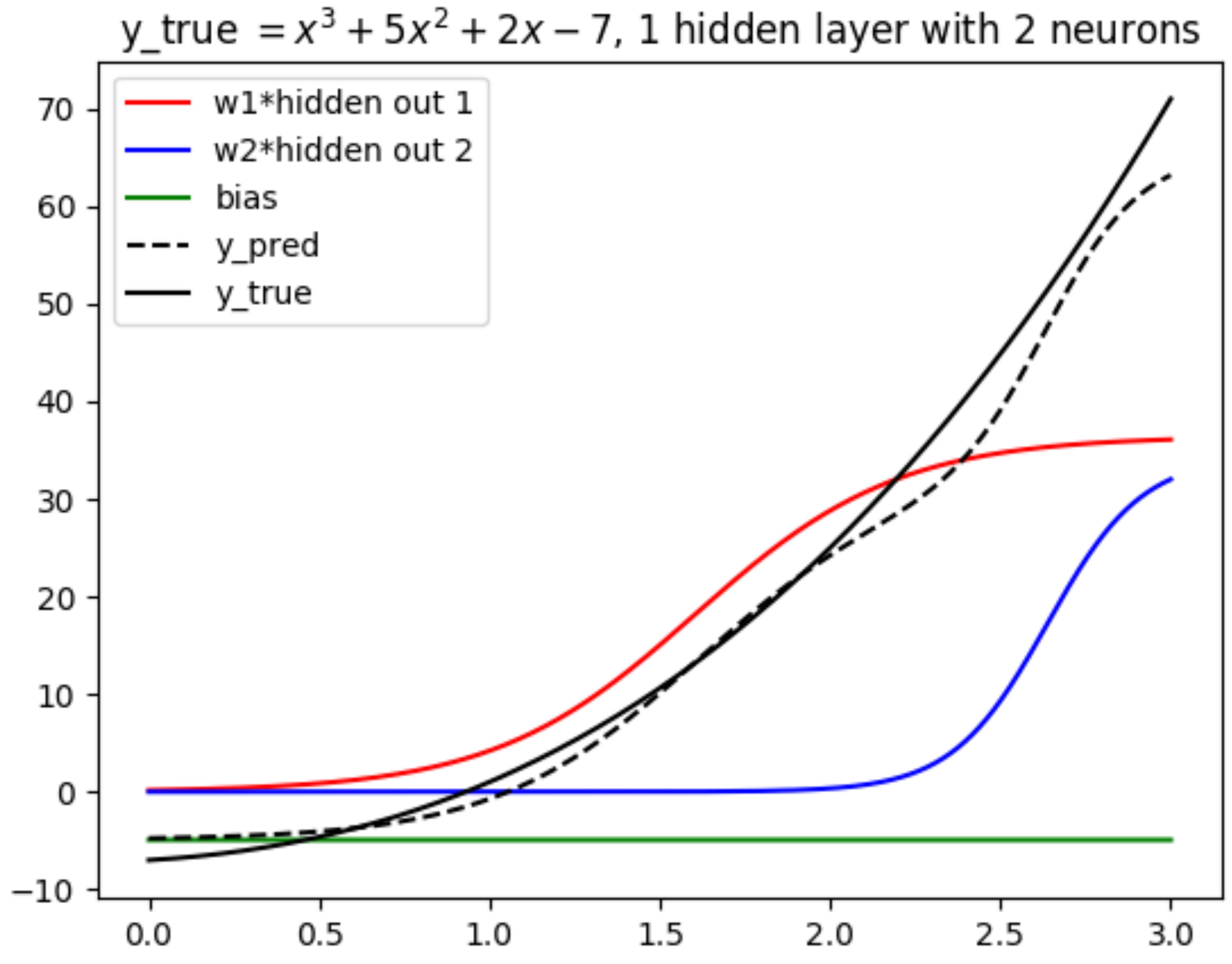


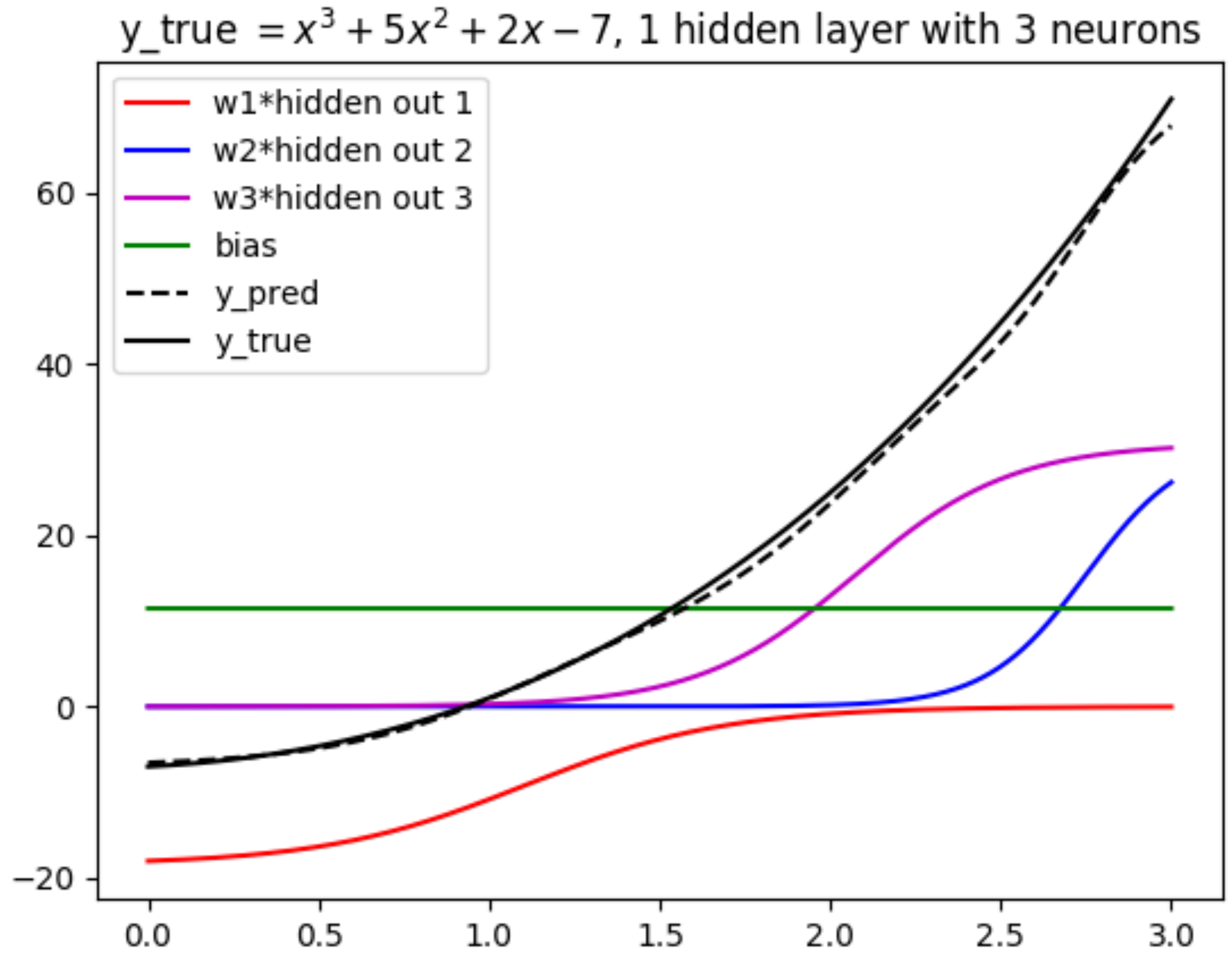
combine bump
functions to get a
Riemann-like
approximation with
many nodes in
hidden layer

Universal Approximation Theorem

What happens when we train a neural net on like this?

http://neuralnetworksanddeeplearning.com/chap4.html





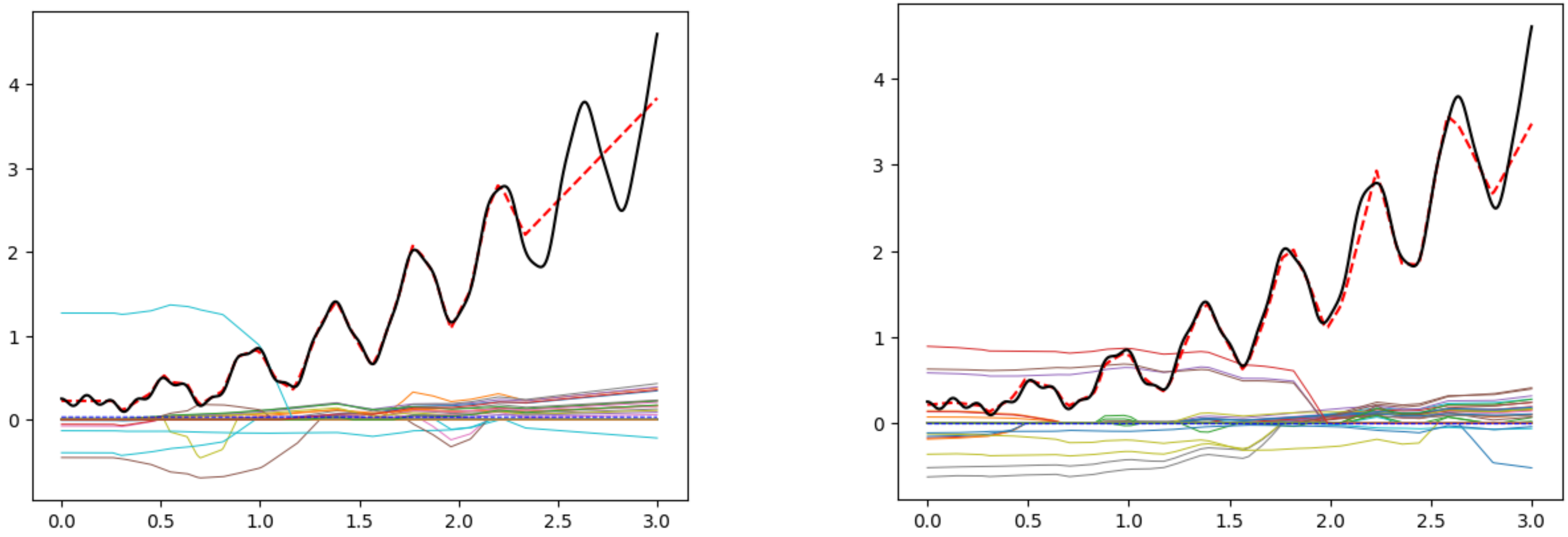
Universal Approximation Theorem

http://neuralnetworksanddeeplearning.com/chap4.html

What happens when we train a neural net on Neilson's crazy 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

dropout (we will see later)

Universal Approximation Theorem

why go deep?

1) single hidden layer may need to be huge

2) not clear that SGD-BP will actually learn this good approximation

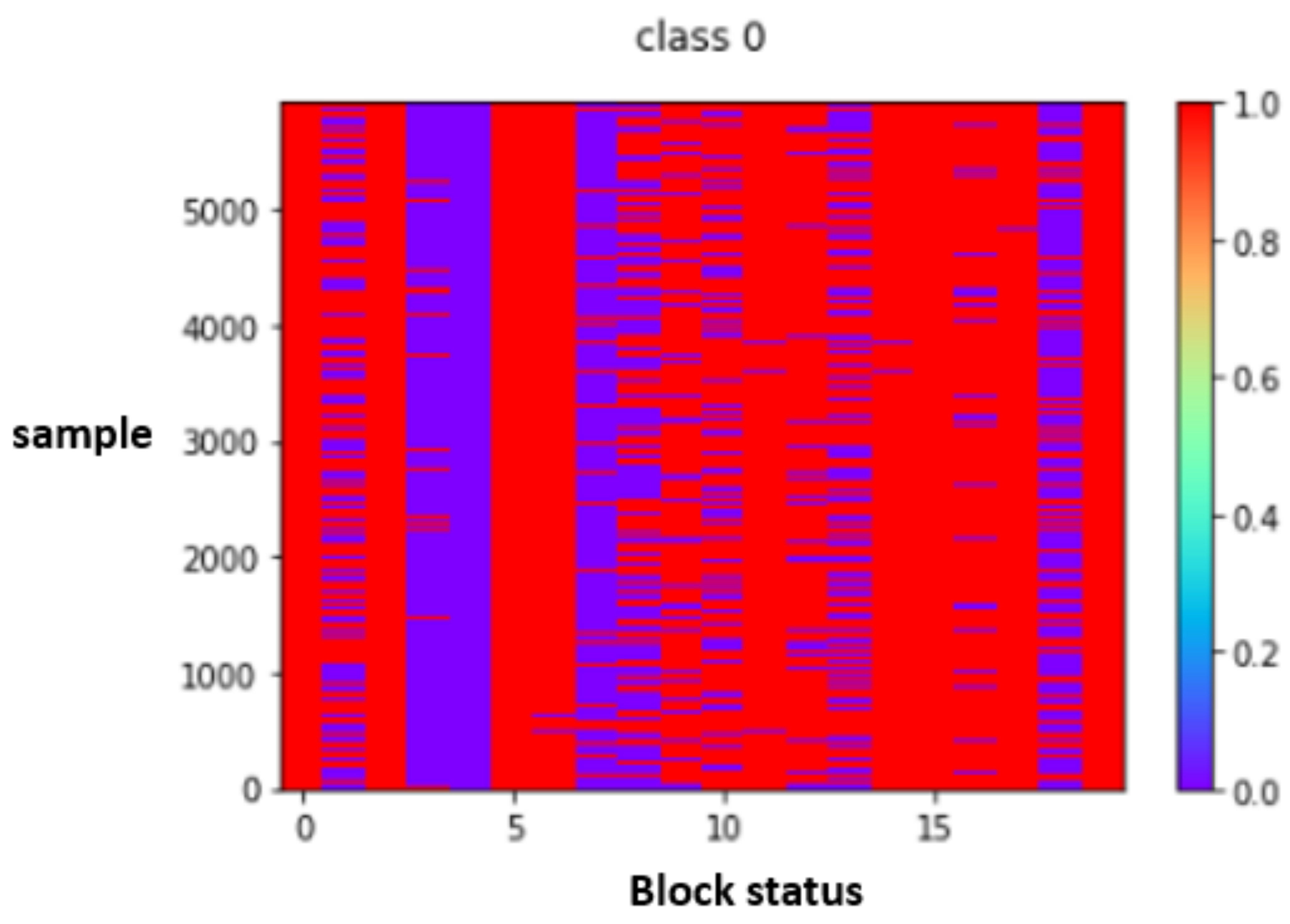
3) There are inherent advantages to more hidden layers

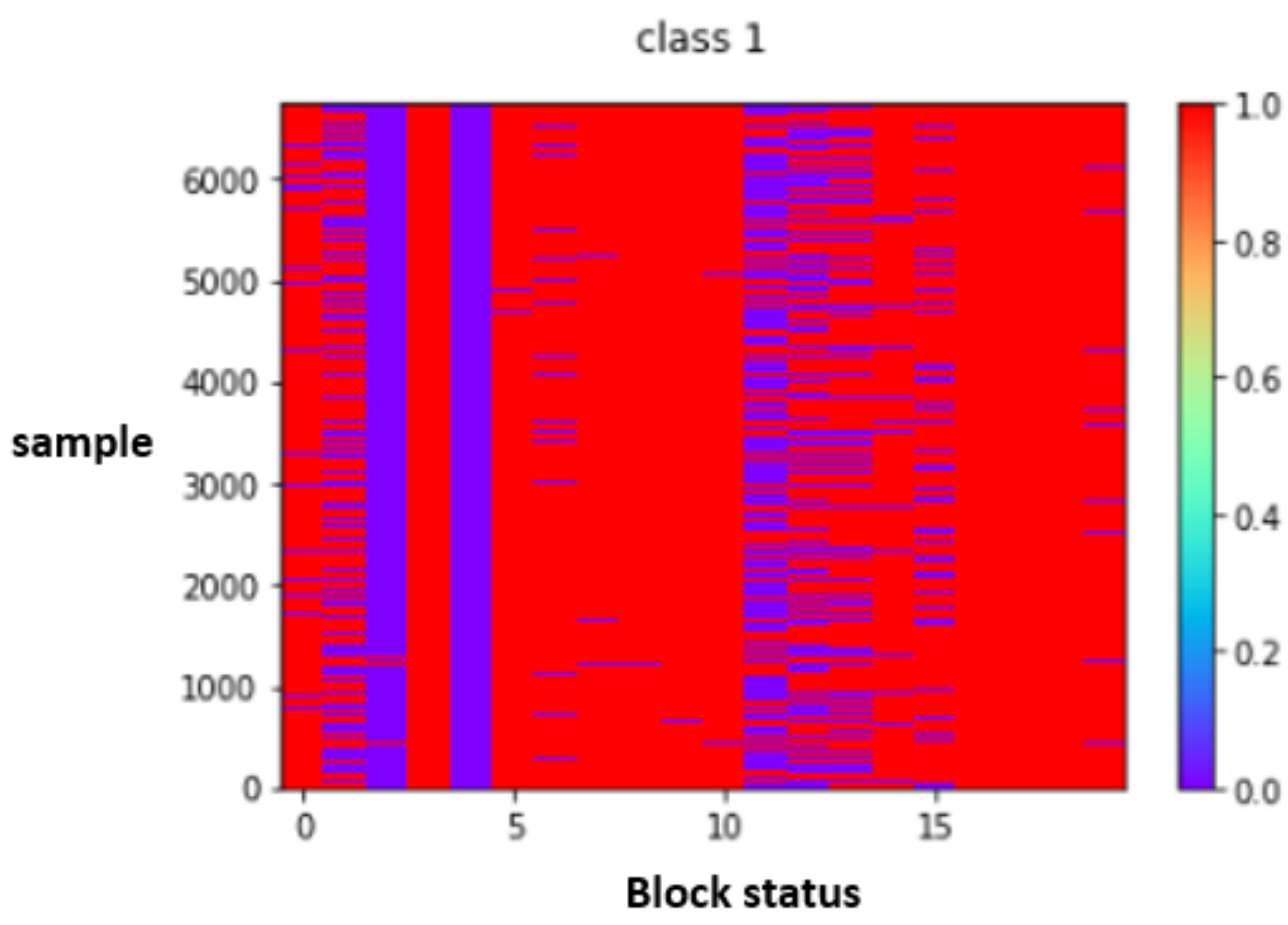
multiple layers can learn stages of classification or "case switches"

e.g.,
Layer1: detect if case A or case B holds
Layer 2: if case A, do algorithm A, else, do algorithm B

many problems suitable to Neural Nets have these properties (I called these "clamps/conditionals" and multiple layers can model this more effectively/efficiently

Example From Class Project (2019)





20 hidden nodes, shows whether rely is ON/OFF for each element in the dataset

can think of a relu-based MLP as configuring switches (classifying) and then applying a linear mapping (these are like the clamps/conditonals)

Conditional Linear Regression: An alternative structure to Deep neural network with ReLU activation
Qianmu Yu, Runmian Chang, Mo Shi

Universal Approximation Theorem

why go deep?

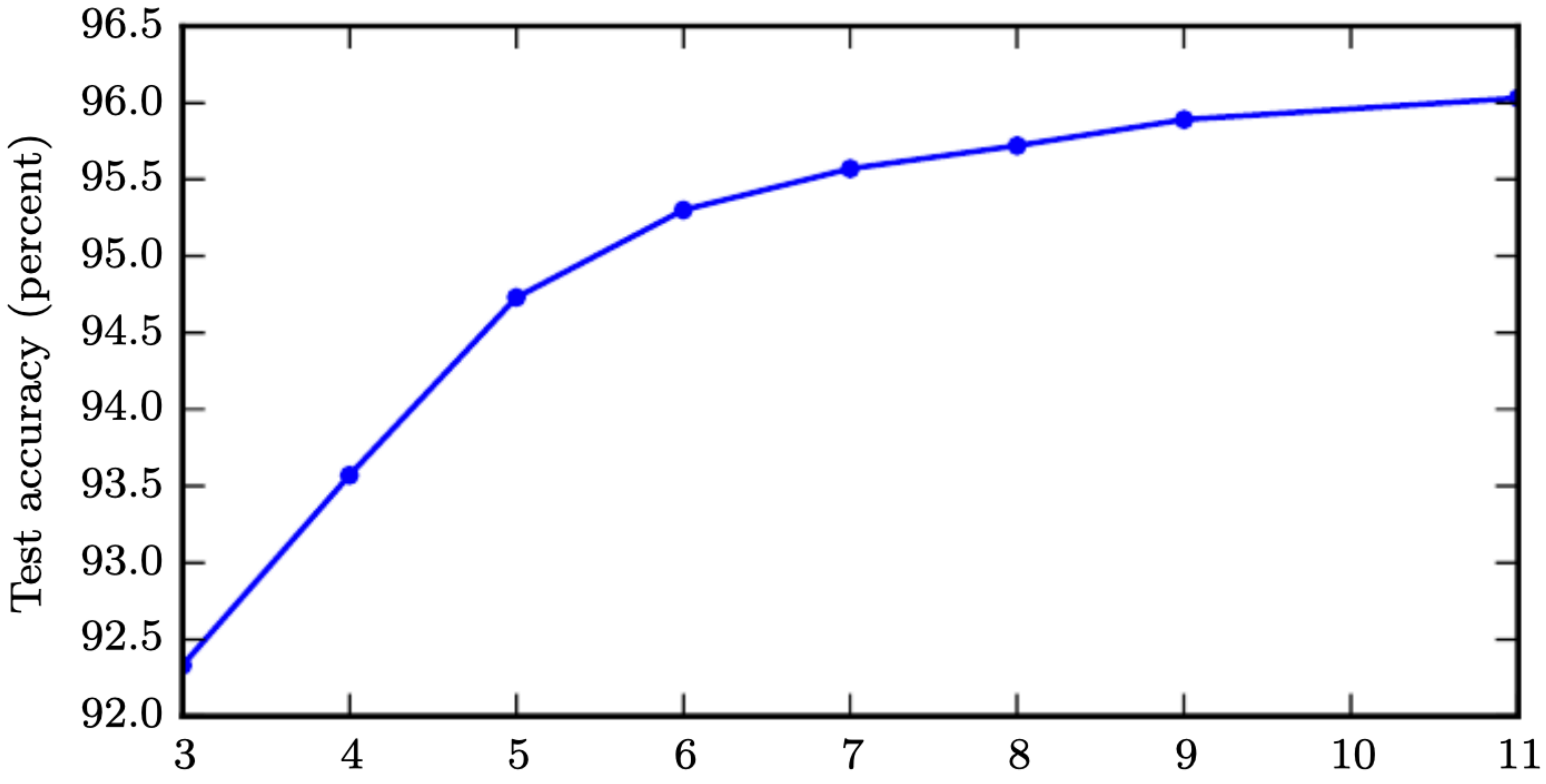


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

deeper models tend to perform better

[GBC] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, The MIT Press, 2016.

Universal Approximation Theorem

why go deep?

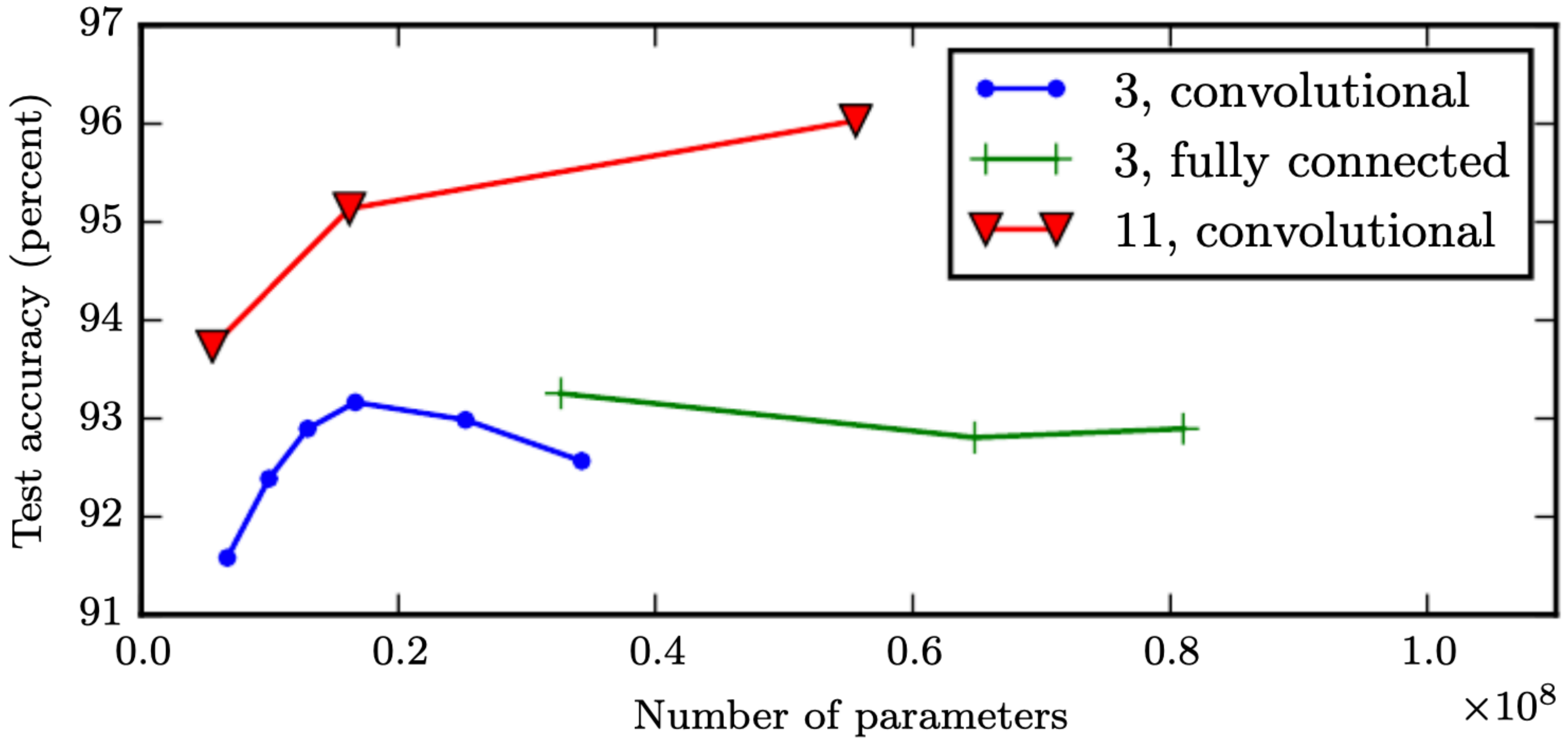


Figure 6.7: Effect of number of parameters. Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance, as illustrated in this figure. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

deeper models tend to perform better

[GBC] Ian Goodfellow, Yoshua Bengio, Aaron Courville, Deep Learning, The MIT Press, 2016.

Outline for Slides

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

Gentle Introduction to tf.keras

TAs will help you install (when ready)

Use tensorflow 2.1 (tf.keras included)

Tensorflow is not part of anaconda…

best to set up virtual-environment in anaconda

(or use pyenv to do minimal virtualenvs and manage easily)

I use: pyenv, **tf 2.1**, macOS, ubuntu 18.0.4, **Python 3.7.4**

Gentle Introduction to tf.keras

Let’s use the “train_fashion_mnist.py” as a starting point

https://github.com/tensorflow/docs/blob/master/site/en/tutorials/keras/classification.ipynb

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
# train_images.shape is (60000, 28, 28)
#test_images.shape (10000, 28, 28)
num_pixels = 28 * 28
train_images = train_images.reshape( (60000, num_pixels) ).astype(np.float32) / 255.0
test_images = test_images.reshape( (10000, num_pixels) ).astype(np.float32)  / 255.0

our_first_model = keras.Sequential([
    keras.layers.Input(shape=(num_pixels,), name='images'),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10, activation='softmax')
])

our_first_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

results = our_first_model.fit(train_images,  train_labels, batch_size=32, epochs=40, validation_split=0.1)

# using a .hdf5 or .h5 extension saves the model in format compatible with older keras
our_first_model.save('fmnist_trained.hdf5')
```

typical import

keras has some standard dataset built in (will download for you)

reshape so that the input is a 1-dim array for an MLP. (done before with a flatten layer)

defines a model using the Sequential method

before training, you need to **compile the model** which tells it what loss and optimizer to use

this does the training

save the model so that you can read it in and use it for inference

Gentle Introduction to tf.keras

Does exactly the same thing, but uses the "Functional API" for defining the model

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
# train_images.shape is (60000, 28, 28)
#test_images.shape (10000, 28, 28)
num_pixels = 28 * 28
train_images = train_images.reshape( (60000, num_pixels) ).astype(np.float32) / 255.0
test_images = test_images.reshape( (10000, num_pixels) ).astype(np.float32)  / 255.0

# this uses the Functional API for definning the model
nnet_inputs = keras.layers.Input(shape=(num_pixels,), name='images')
z = keras.layers.Dense(128, activation='relu', name='hidden')(nnet_inputs)
z = keras.layers.Dense(10, activation='softmax', name='output')(z)

our_first_model = keras.Model(inputs=nnet_inputs, outputs=z)


our_first_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
results = our_first_model.fit(train_images,  train_labels, batch_size=32, epochs=40, validation_split=0.1)

# using a .hdf5 or .h5 extension saves the model in format compatible with older keras
our_first_model.save('fmnist_trained.hdf5')
```

defines a model using the Functional API method

(layer name is optional, but good practice)

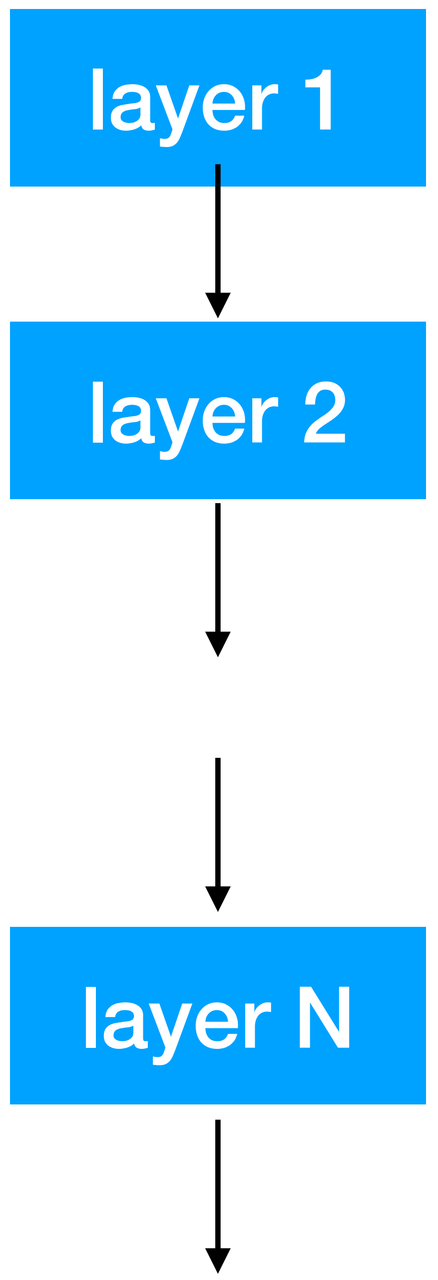
tf.keras — defining the model

Sequential

simple, quick

not very flexible

only allows for models that are a sequence of layers (line-graph)



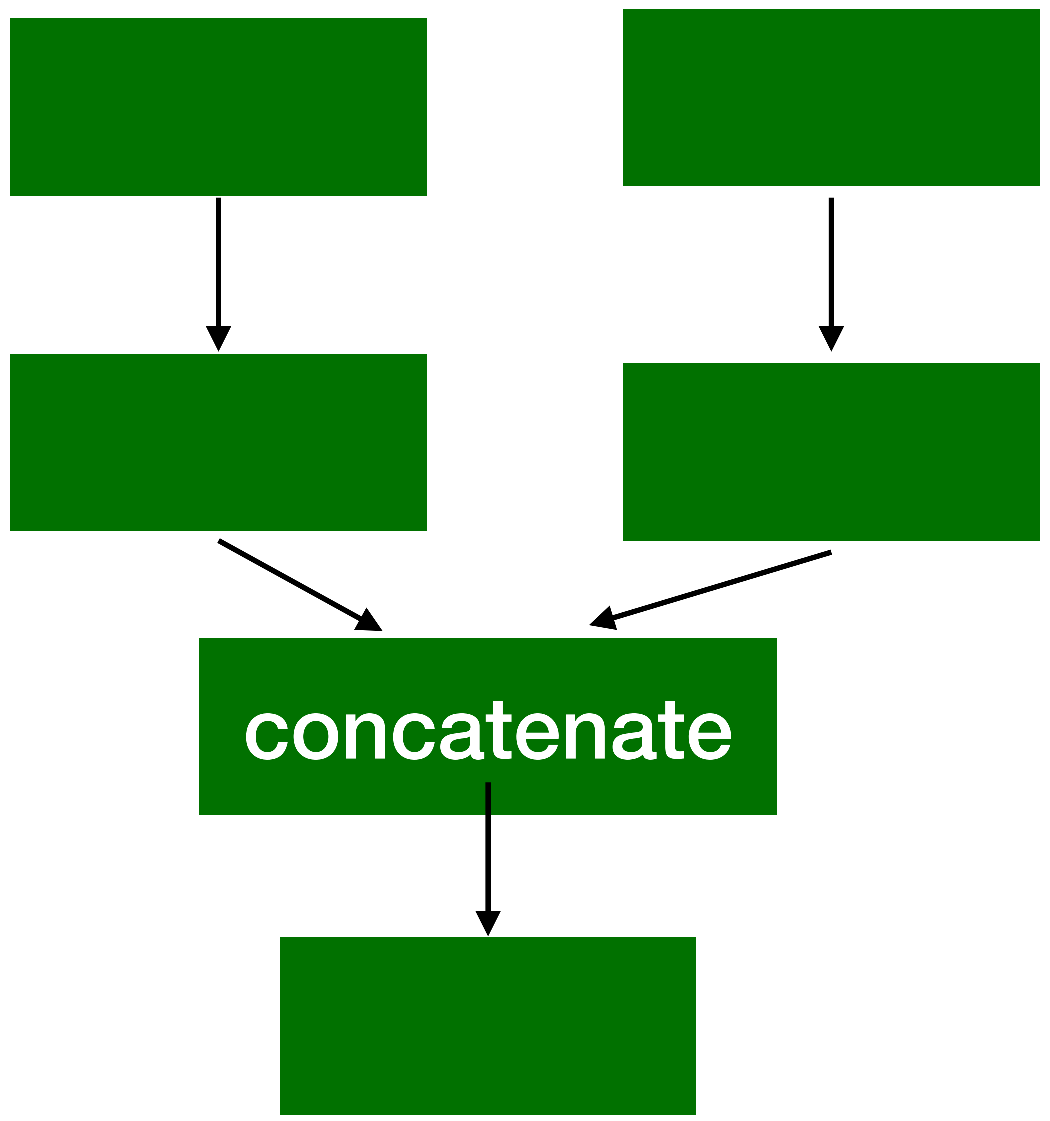
I only use the Functional API and recommend you use it too

Functional API

maybe a little more work?

much more powerful:

- Models with shared layers
- Multi-input, multi-output models
- Directed acyclic graphs (DAGs)
- Custom layer
- Custom function on intermediate layer's output



tf.keras — viewing model structure

```
our_first_model = keras.Model(inputs=nnet_inputs, outputs=z)

#this will print a summary of the model to the screen
our_first_model.summary()

#this will produce a digram of the model -- requires pydot and graphviz installed
keras.utils.plot_model(our_first_model, to_file='our_first_model.png', show_shapes=True, show_layer_names=True)

our_first_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
results = our_first_model.fit(train_images,  train_labels, batch_size=32, epochs=40, validation_split=0.1)
```

pydot and graphviz are utilities for plotting block diagrams and graphs

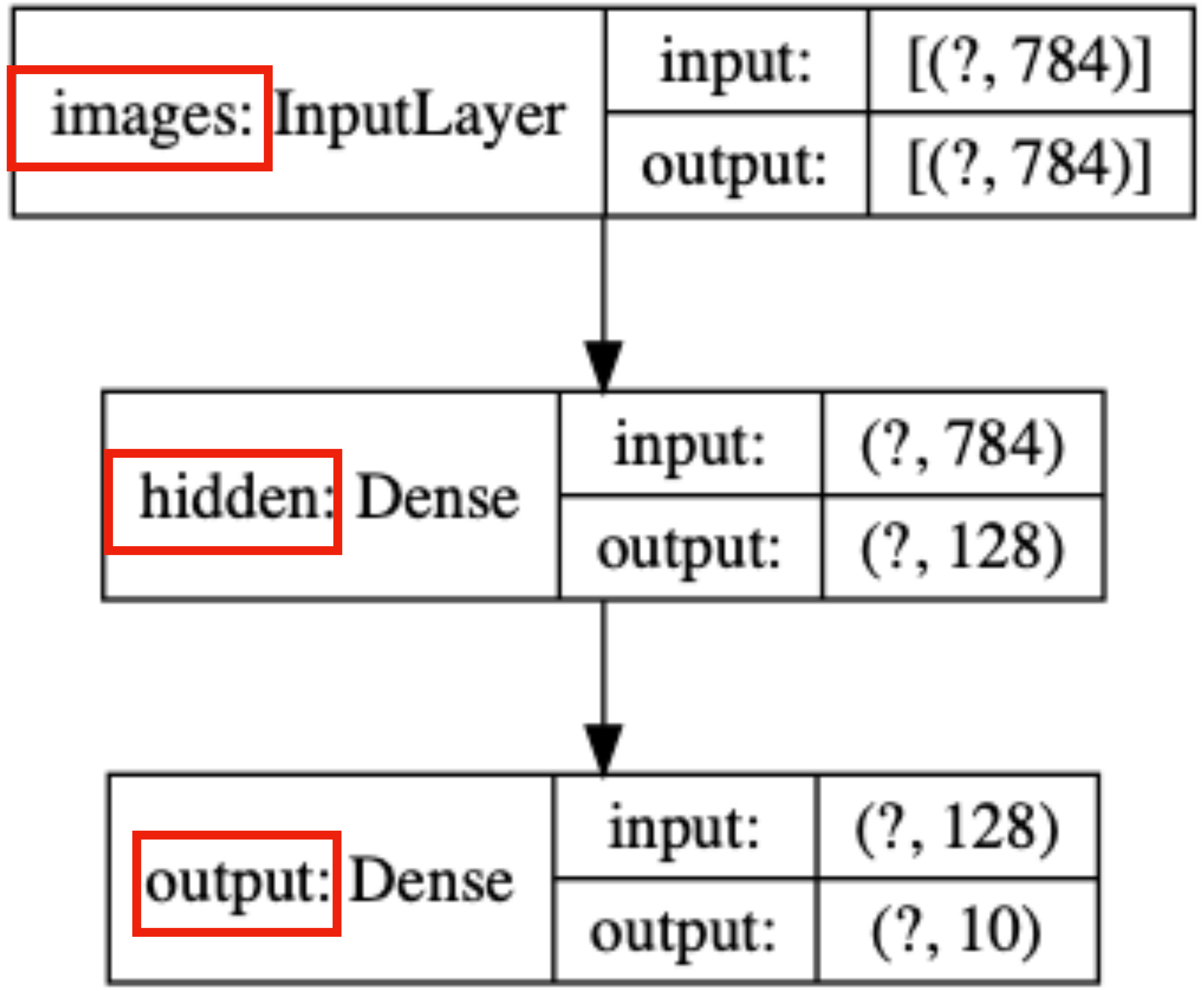
tf.keras — viewing model structure

model summary prints out the layer shapes and number of trainable parameters

```
train21 〉 ~/Documents/USC/classes/EE599_deep_learning/sp2020-ee599/example_scripts/lecture_examples/fashion_mnist 〉 master ● ? 〉 python 3_fmnist.py
020-02-11 18:02:40.857273: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use
020-02-11 18:02:40.916070: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x7fd7941d73b0 initialized for platform Host (this does not guarantee tha
ces:
020-02-11 18:02:40.916099: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default Version
odel: "model"
_________________________________________________________________
ayer (type)                  Output Shape              Param #
=================================================================
mages (InputLayer)           [(None, 784)]             0
_________________________________________________________________
idden (Dense)                (None, 128)               100480
_________________________________________________________________
utput (Dense)                (None, 10)                1290
=================================================================
otal params: 101,770
rainable params: 101,770
on-trainable params: 0
_________________________________________________________________
rain on 54000 samples, validate on 6000 samples
poch 1/40
6288/54000 [===================>.........] - ETA: 3s - loss: 0.5491 - accuracy: 0.8109
```

tf.keras — viewing model structure

plot_model() produces this diagram



shows the names we gave to the layers

the ? is there because we did not specify the batch size when defining the model.

will work with any batch size.

tf.keras — checking performance

the model.fit returns a dictionary that has all of the train/val losses

```
>>> results.history.keys()
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

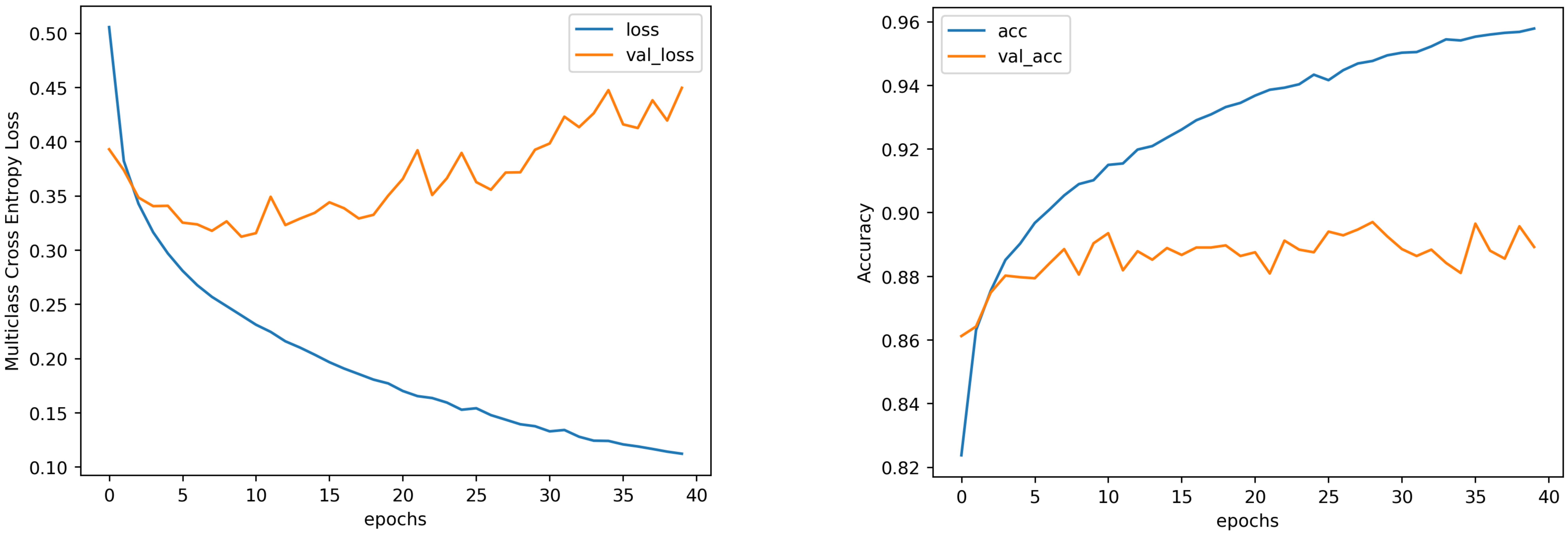
```
# plot our learning curves
#results.history is a dictionary
loss = results.history['loss']
val_loss = results.history['val_loss']
acc = results.history['accuracy']
val_acc = results.history['val_accuracy']

epochs = np.arange(len(loss))

plt.figure()
plt.plot(epochs, loss, label='loss')
plt.plot(epochs, val_loss, label='val_loss')
plt.xlabel('epochs')
plt.ylabel('Multiclass Cross Entropy Loss')
plt.legend()
plt.savefig('learning_loss.png', dpi=256)

plt.figure()
plt.plot(epochs, acc, label='acc')
plt.plot(epochs, val_acc, label='val_acc')
plt.xlabel('epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('learning_acc.png', dpi=256)
```

each of these is a numpy array

just standard plotting

tf.keras — checking performance

results of our training run…



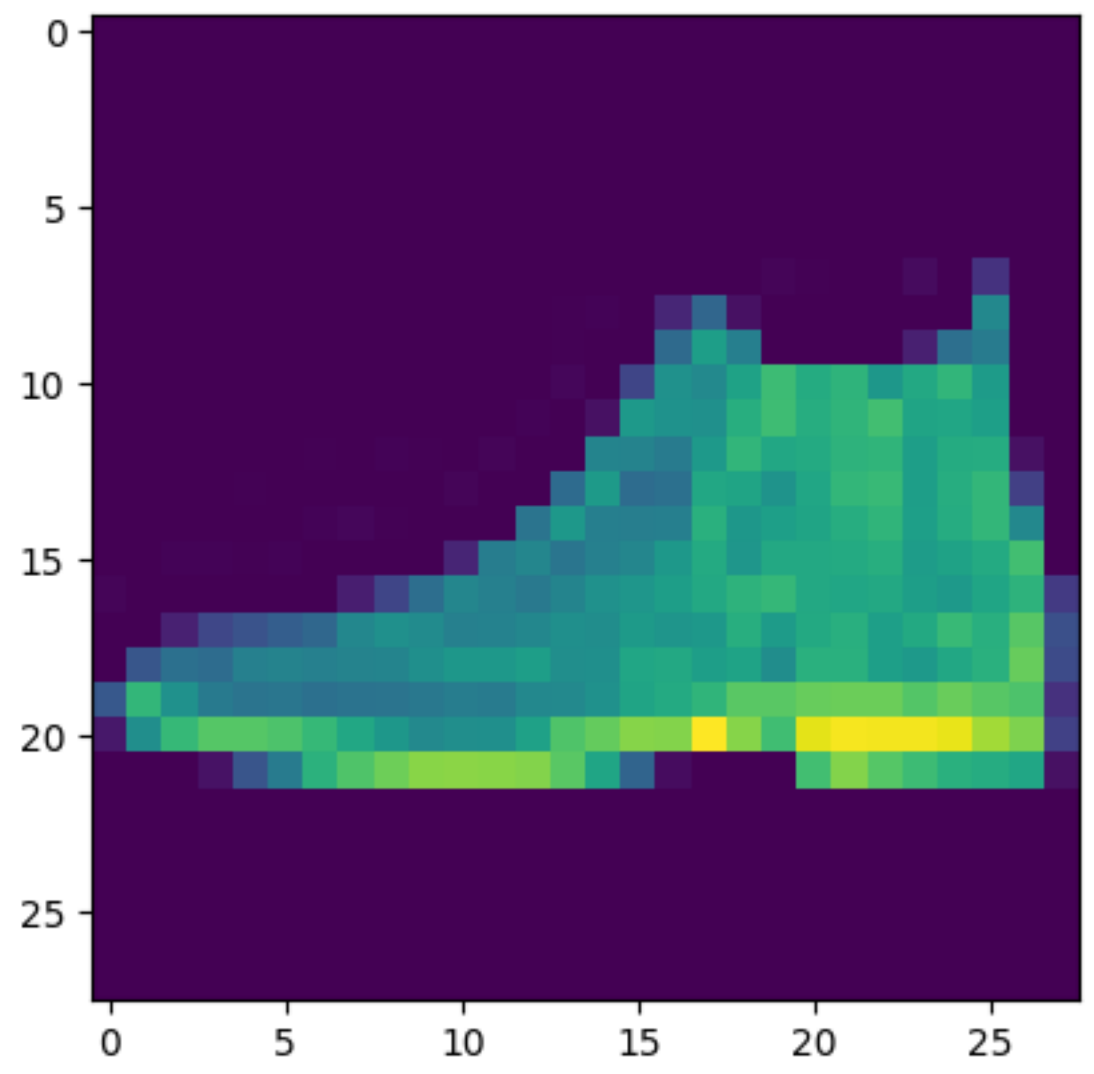
over-fitting (bad!)

tf.keras — checking performance

let's try running inference on an image…

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

```
>>> plt.imshow(test_images[0].reshape(28,28))
```



the first test image is an Ankle Boot (class 9)

tf.keras — checking performance

let's try running inference on an image…

```
# read back out model, just to illustrate
model_copy = keras.models.load_model('fmnist_trained.hdf5')

# perform inference on a single image:
prediction = model_copy.predict(test_images[0].reshape(1,num_pixels))
num_classes = 10
prediction = prediction.reshape(10)
class_decision = np.argmax(prediction)
for m in range(num_classes):
    if m == class_decision:
        print(f'class{m}:\tclass soft-decisions:{prediction[m]}\t(hard decision)')
    else:
        print(f'class{m}:\tclass soft-decisions:{prediction[m]}')
```

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

need to reshape the input to the network so it has shape:
(prediction_batch_size, input_shape)

reshape the output because it also returns a multi-dimensional tensor (has batch dimension)

tf.keras — checking performance

let's try running inference on an image…

```
class0: class soft-decisions:2.6553041449633996e-12
class1: class soft-decisions:6.97358803014681e-19
class2: class soft-decisions:3.6388213541524786e-14
class3: class soft-decisions:4.071454019874453e-15
class4: class soft-decisions:1.663703490294502e-14
class5: class soft-decisions:1.0153004950552713e-05
class6: class soft-decisions:1.480168378975577e-07
class7: class soft-decisions:0.0003396416432224214
class8: class soft-decisions:1.170382454146468e-11
class9: class soft-decisions:0.9996500015258789 (hard decision)
```

Label	Class
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Yeah! It worked on that one (despite the over fitting)

You can pass many images to model.predict (batch >1)
and it will return all of the "predictions"

tf.keras — checking performance

Use model.evaluate to get the loss and metrics for the test set…

```
test_loss, test_acc = model_copy.evaluate(test_images,  test_labels, verbose=2)
print(f'Test Loss: {test_loss : 3.2f}')
print(f'Test Accuracy: {100 * test_acc : 3.2f}%')
```

result:

Test Loss: 0.50
Test Accuracy: 88.44%

Note that this is very similar to the performance on the validation set

What’s left to know about tf.keras?

- Options… that is next — learn the ideas and show how done in tf.keras.
- Custom callbacks
  - Using tensorflow.keras.callbacks.Callback class and methods
  - Can save (best) model at epoch end, plot learning curves, etc.
- Custom Layers and Losses
- Dataloaders — can’t fit the entire dataset in RAM…
  - Using tensorflow.keras.utils.Sequence class and methods
- Tensorboard (if you want…)

Outline for Slides

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

Vanishing Gradient Problem

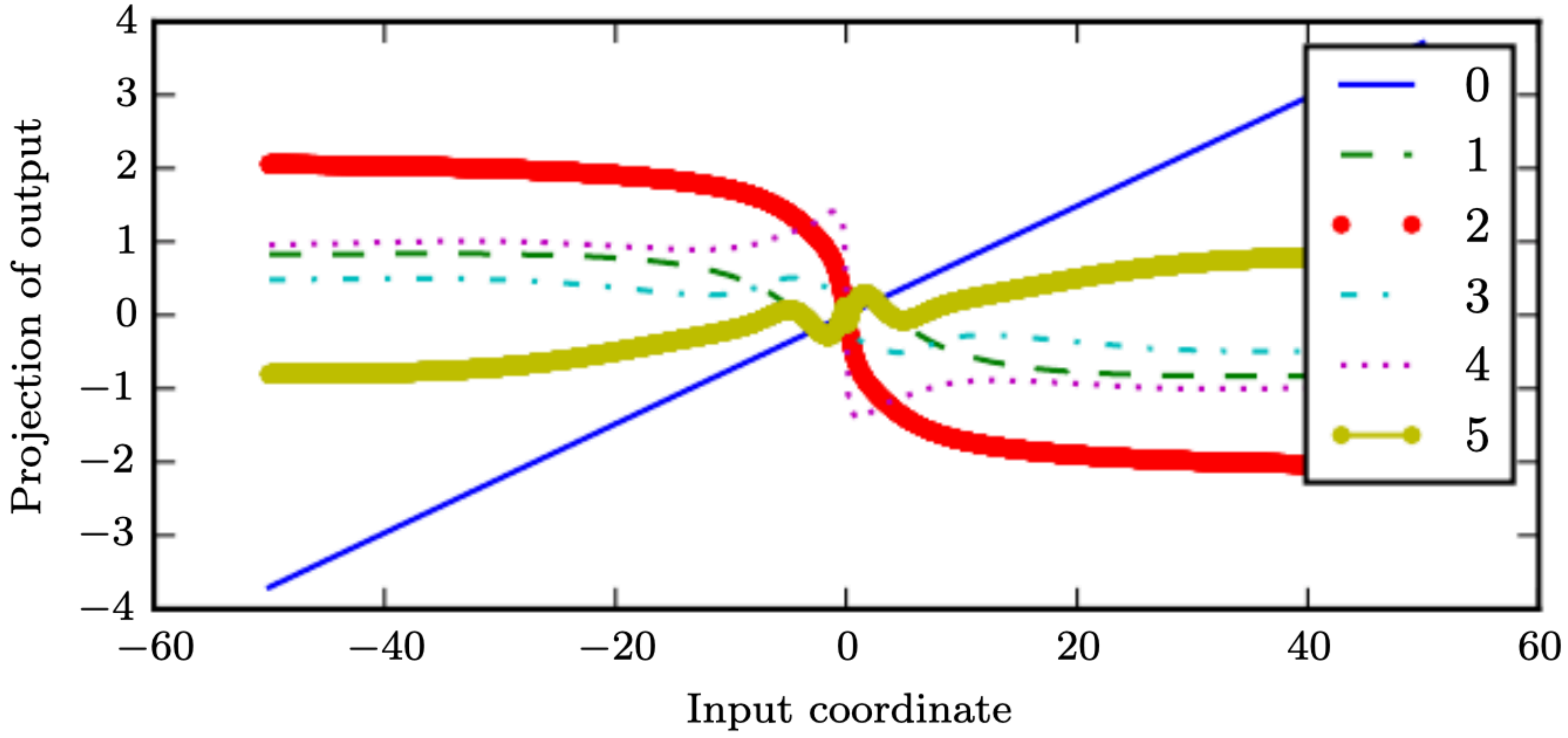


Figure 10.15: Repeated function composition. When composing many nonlinear functions (like the linear-tanh layer shown here), the result is highly nonlinear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many alternations between increasing and decreasing. Here, we plot a linear projection of a 100-dimensional hidden state down to a single dimension, plotted on the y-axis. The x-axis is the coordinate of the initial state along a random direction in the 100-dimensional space. We can thus view this plot as a linear cross-section of a high-dimensional function. The plots show the function after each time step, or equivalently, after each number of times the transition function has been composed. [GBC - Deep Learning]

the gradient can get small as we back-prop

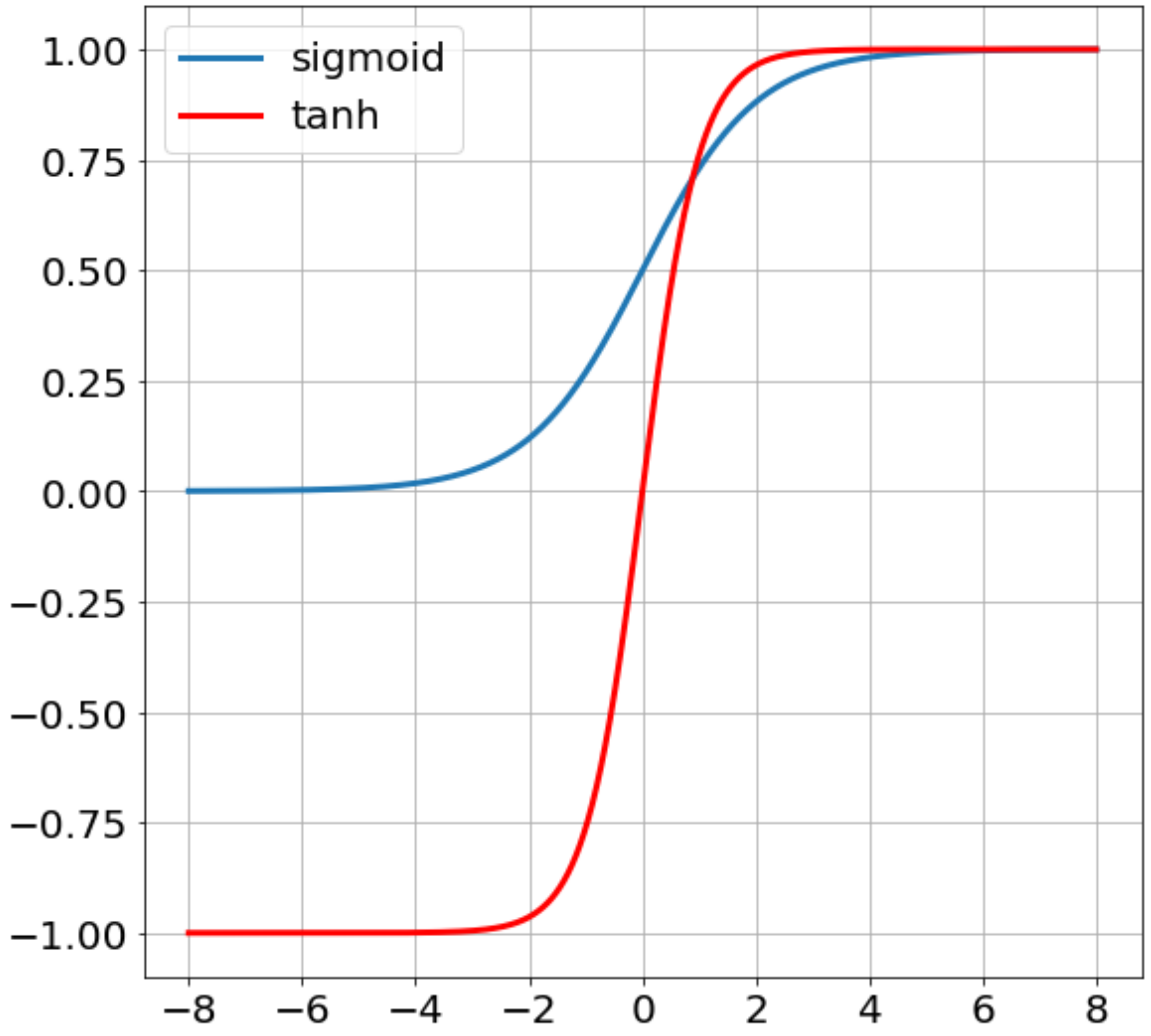
due to the squashing activation compounded effects

See section 10.7 of Deep Learning book for further discussion

Vanishing Gradient Problem - Squashing Activations

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

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$= 2\sigma(2x) - 1$$



the gradient can get small
as we back-prop

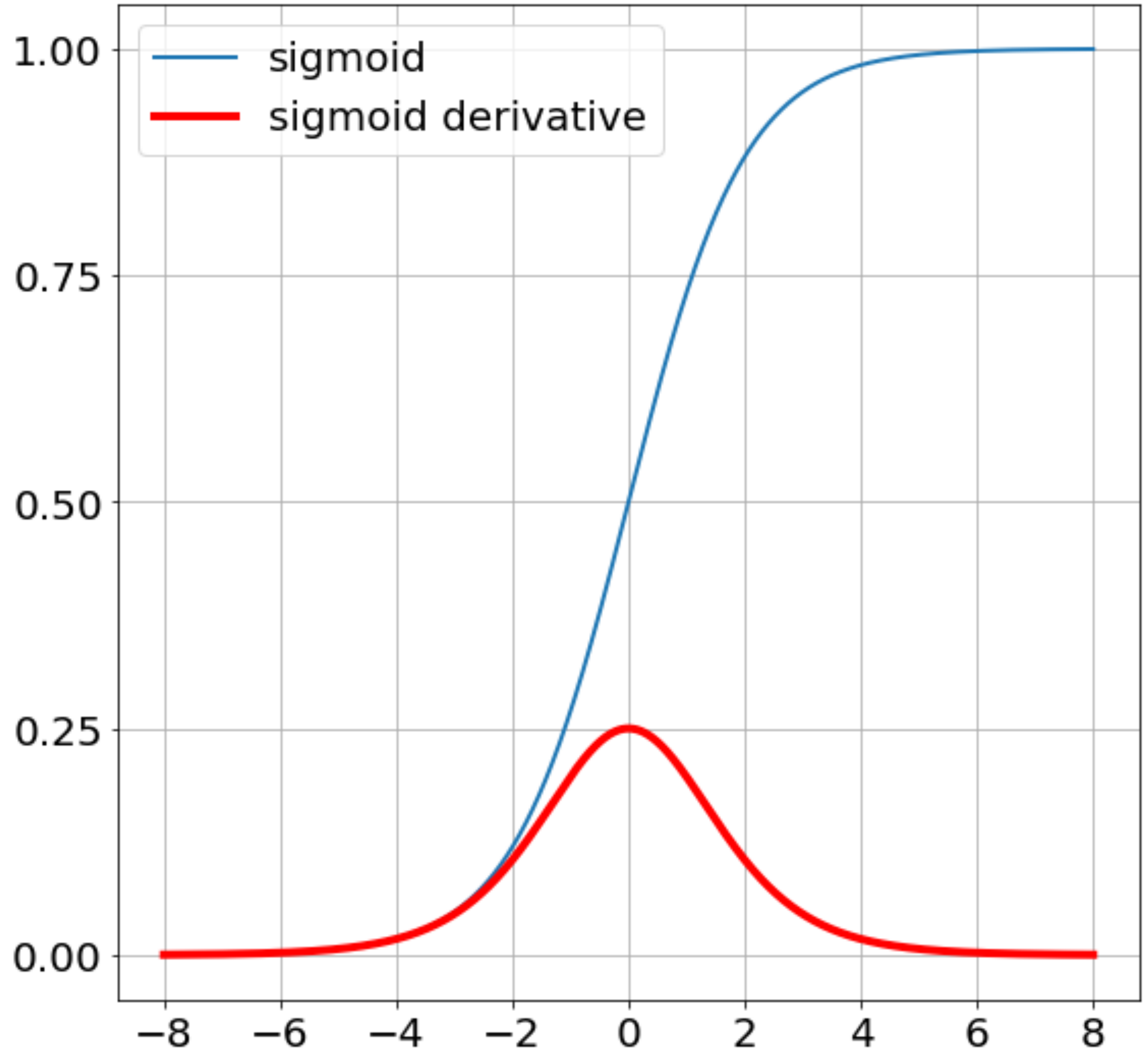
due to the squashing activation
compounded effects

contributions from Sourya Dey

Vanishing Gradient Problem - Squashing Activations

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$$

the maximum value of sigma(.)
is 0.25…



$$\boldsymbol{\delta}_1 = \left(\dot{\sigma}(\mathbf{s}_1) \odot \left[\mathbf{W}_2^{\mathrm{t}}\boldsymbol{\delta}_2\right]\right) \left(\dot{\sigma}(\mathbf{s}_2) \odot \left[\mathbf{W}_3^{\mathrm{t}}\boldsymbol{\delta}_3\right]\right) \left(\dot{\sigma}(\mathbf{s}_3) \odot \left[\mathbf{W}_4^{\mathrm{t}}\boldsymbol{\delta}_4\right]\right) \left(\dot{\sigma}(\mathbf{s}_4) \odot \left[\mathbf{W}_5^{\mathrm{t}}\boldsymbol{\delta}_5\right]\right) \left(\dot{C}(\mathbf{y}, \mathbf{a}_5) \odot \dot{\sigma}(\mathbf{s}_5)\right)$$

contributions from Sourya Dey

Vanishing Gradient Problem - ReLu Activations

Biologically inspired - *neurons firing vs not firing*

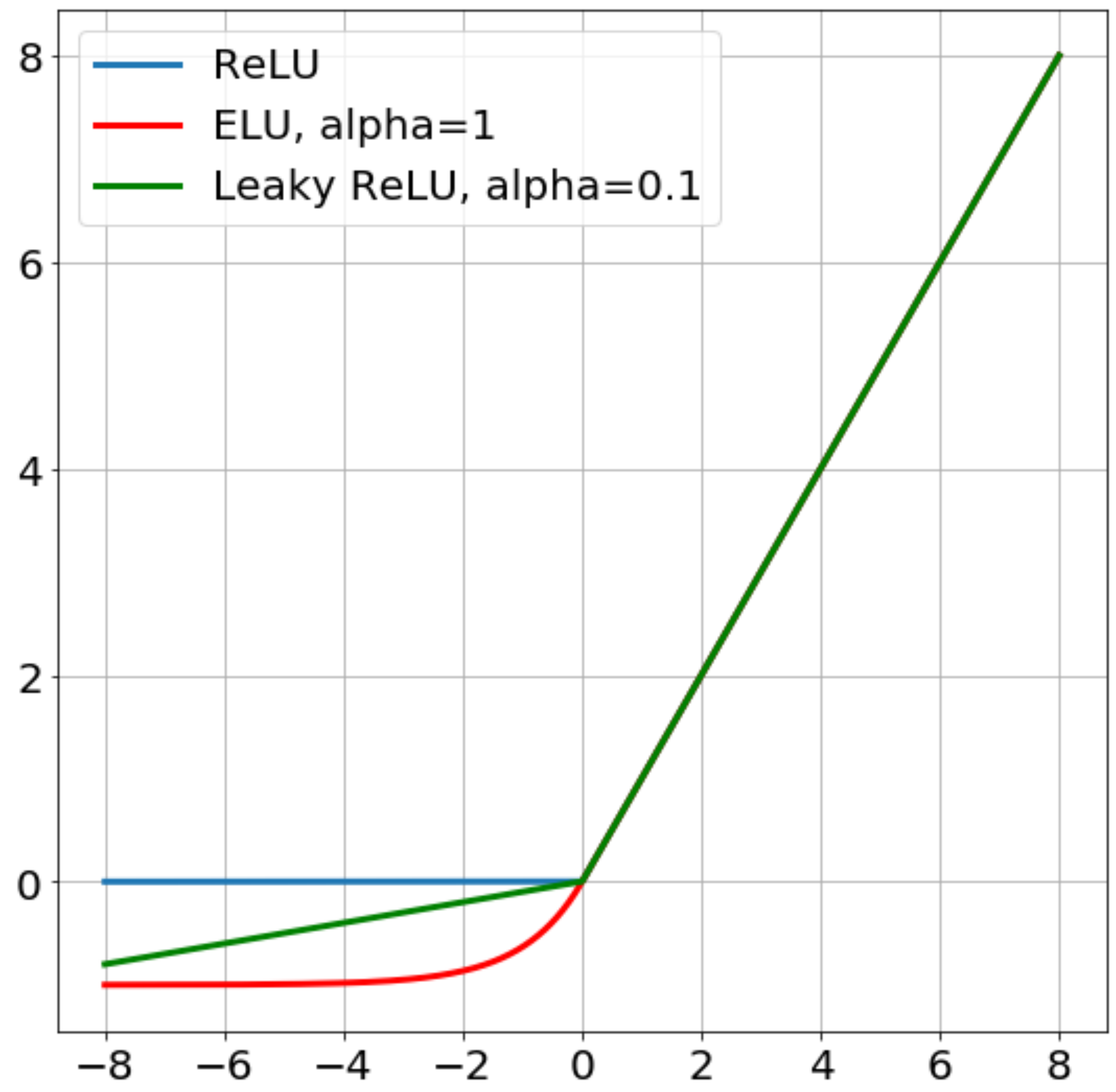
Solves vanishing gradient problem

Non-differentiable at 0, replace with anything in [0,1]

ReLU can die if x<0

Leaky ReLU solves this, but inconsistent results

ELU saturates for x<0, so less resistant to noise



Clevert, Djork-Arné; Unterthiner, Thomas; Hochreiter, Sepp (2015-11-23). "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". arXiv:1511.07289

contributions from Sourya Dey

Activations in tf.keras

```
# this uses the Functional API for definning the model
nnet_inputs = keras.layers.Input(shape=(num_pixels,), name='images')
z = keras.layers.Dense(128, activation='relu', name='hidden')(nnet_inputs)
z = keras.layers.Dense(10, activation='softmax', name='output')(z)
```

https://keras.io/activations/

https://www.tensorflow.org/api_docs/python/tf/keras/activations

Activations in tf.keras

Functions

`deserialize(...)` : Returns activation function denoted by input string.

`elu(...)` : Exponential linear unit.

`exponential(...)` : Exponential activation function.

`get(...)` : Returns function.

`hard_sigmoid(...)` : Hard sigmoid activation function.

`linear(...)` : Linear activation function.

`relu(...)` : Applies the rectified linear unit activation function.

`selu(...)` : Scaled Exponential Linear Unit (SELU).

`serialize(...)` : Returns name attribute (`__name__`) of function.

`sigmoid(...)` : Sigmoid activation function.

`softmax(...)` : Softmax converts a real vector to a vector of categorical probabilities.

`softplus(...)` : Softplus activation function.

`softsign(...)` : Softsign activation function.

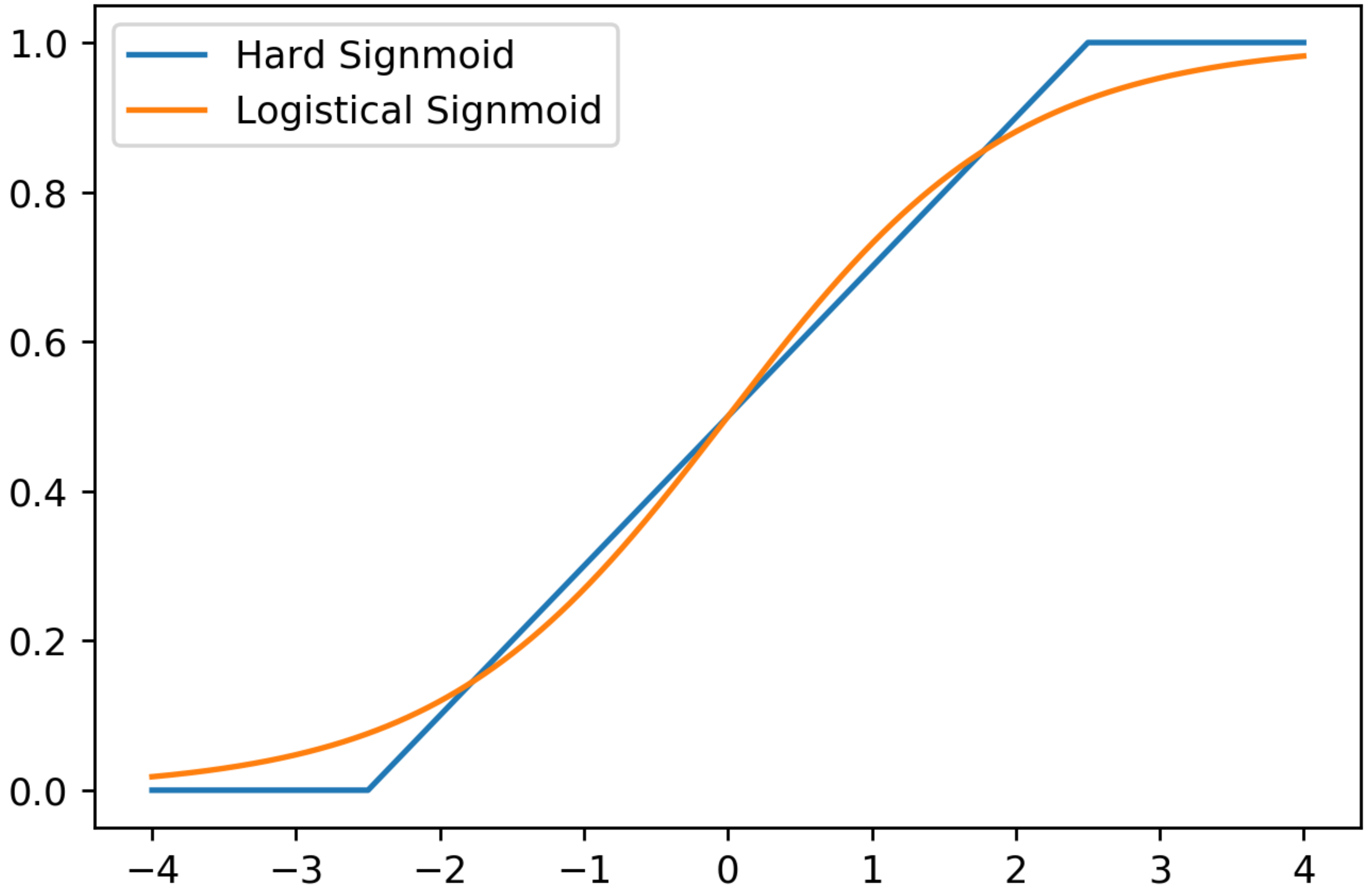
`tanh(...)` : Hyperbolic tangent activation function.

https://www.tensorflow.org/api_docs/python/tf/keras/activations

layers have a default activations in tf.keras…

dense, convolutional have linear as default
RNNs use, tanh, sigmoid, hard_sigmoid depending on variant

Activations in tf.keras




```python
def hard_sigmoid(x):
    if np.abs(x) < 2.5:
        y = 0.2 * x + 0.5
    elif x > 0:
        y = 1
    elif x < 0:
        y = 0
    return y
```

hard_sigmoid sometimes used to reduce computation

Activations in tf.keras

$$\mathbf{h}(\mathbf{s}) = \frac{1}{\sum_{m=0}^{M-1} e^{s_m}} \begin{bmatrix} e^{s_0} \\ e^{s_1} \\ \vdots \\ e^{s_{M-1}} \end{bmatrix}$$

soft-max:
produces M x 1 probability mass function use for M-ary classification between mutual exclusive classes

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

sigmoid:
produces probability of “class 1” for a binary classification test

binary classification:
1 output neuron with sigmoid and BCE
vs.
2 output neurons with softmax and MCE

Outline for Slides

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

Weight (and bias) Initialization

$$\theta \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}$$

what do we initialize parameter theta with?

empirical observation: some initializations are better than others

zero initialization?
all linear activations are 0…
the detas will be 0 too…

$$\boldsymbol{\delta}_l = \dot{\mathbf{a}}_l \odot \left[\mathbf{W}_{l+1}^{\mathrm{t}} \boldsymbol{\delta}_{l+1} \right]$$

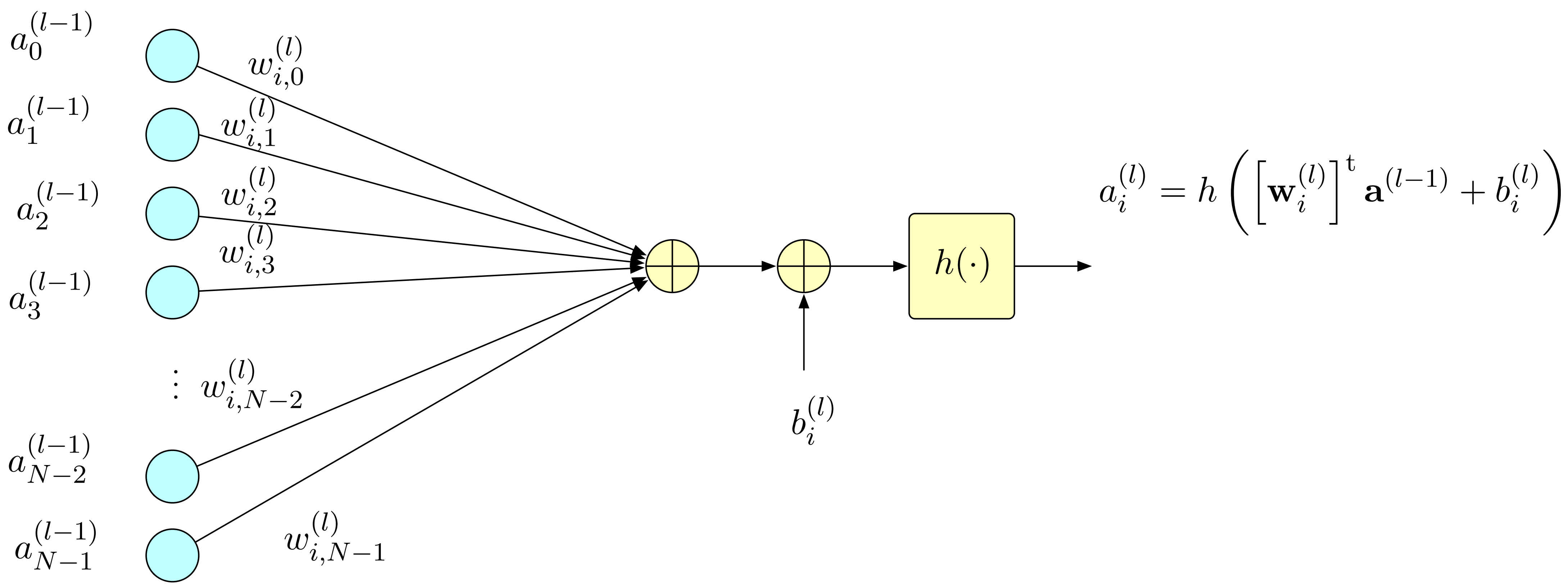
use random initialization…

Weight (and bias) Initialization

Glorot (Xavier) Normal Initialization

Consider a linear function:
assume all w, x are IID:

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_N x_N$$
$$\mathrm{Var}(y) = N\mathrm{Var}(w)\mathrm{Var}(x)$$
$$\text{if } \mathrm{Var}(w) = \frac{1}{N}$$
$$\text{then } \mathrm{Var}(y) = \mathrm{Var}(x)$$



This suggests:

$$\text{Feedforward:} \quad \sigma^2_{w_{i,j}^{(l)}} \approx \frac{1}{N_{l-1}}$$
$$\text{Backprop:} \quad \sigma^2_{w_{i,j}^{(l)}} \approx \frac{1}{N_l}$$

$$\boxed{w_{i,j}^{(l)} \sim \mathcal{N}\left(0; \frac{2}{N_{l-1} + N_l}\right)}$$

Xavier Glorot, Yoshua Bengio. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010.

contributions from Sourya Dey

Weight (and bias) Initialization

Glorot (Xavier) Uniform Initialization

use same second moments with uniform initialization….

$$w_{i,j}^{(l)} \sim \text{uniform}(-a, +a)$$

$$\sigma^2_{w_{i,j}^{(l)}} = \frac{a^2}{3}$$

$$\sigma^2_{w_{i,j}^{(l)}} = \frac{2}{N_{l-1} + N_l}$$

$$a = \sqrt{\frac{6}{N_{l-1} + N_l}}$$

$$\boxed{w_{i,j}^{(l)} \sim \text{uniform}\left(-\sqrt{\frac{6}{N_{l-1} + N_l}}, +\sqrt{\frac{6}{N_{l-1} + N_l}}\right)}$$

Xavier Glorot, Yoshua Bengio. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR 9:249-256, 2010.

Weight (and bias) Initialization

He Initailization

Glorot does not account for nonlinear activations (e.g., ReLU)

$$w_{i,j}^{(l)} \sim \mathcal{N}\left(0; \frac{2}{N_{l-1}}\right)$$

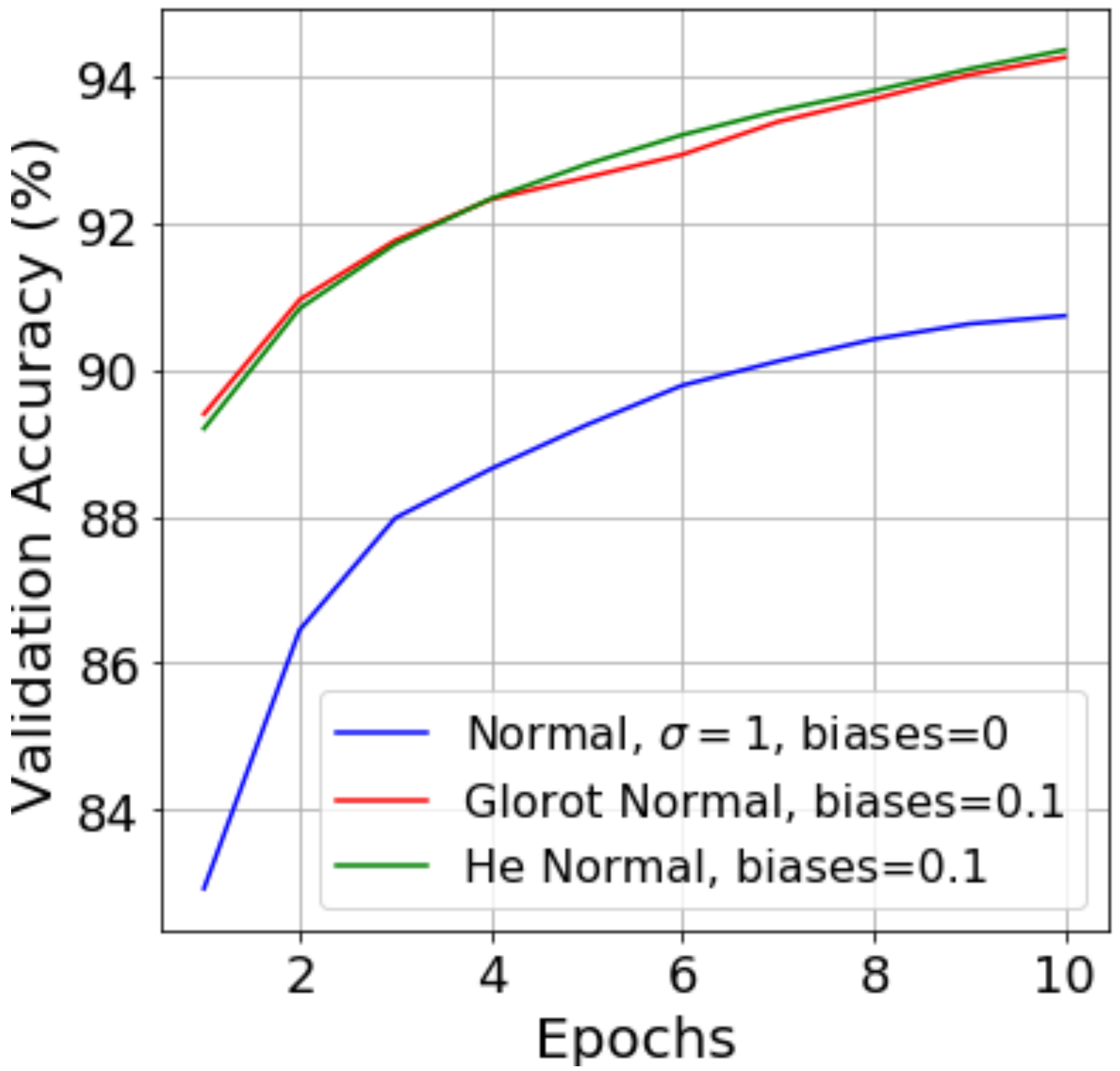
He Normal Initialization

$$w_{i,j}^{(l)} \sim \text{uniform}\left(-\sqrt{\frac{6}{N_{l-1}}}, +\sqrt{\frac{6}{N_{l-1}}}\right)$$

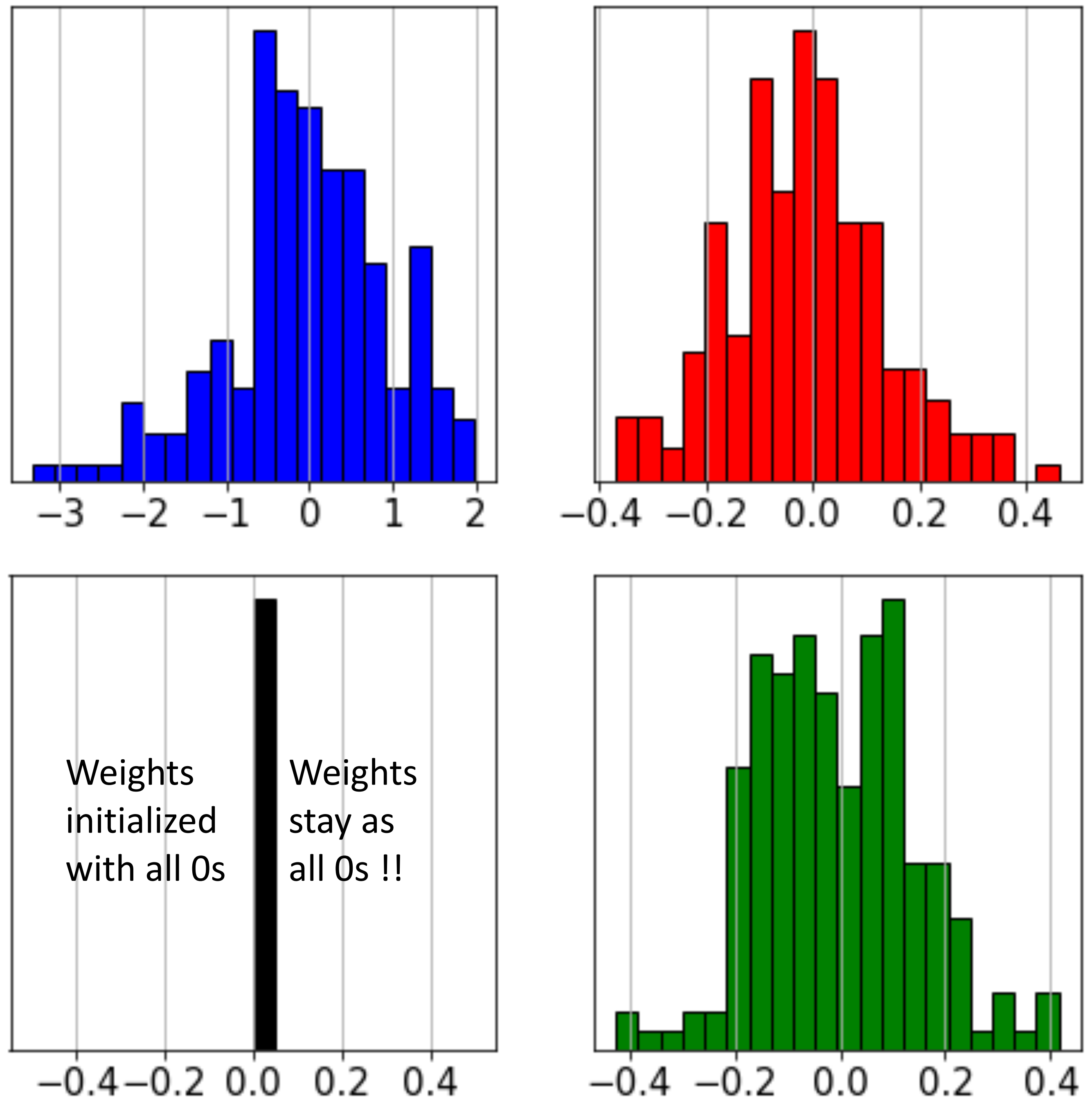
He Uniform Initialization

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. Proceedings of ICCV '15, pp 1026-1034.

Comparison of Initializers



MNIST [784,200,10]
Regularization: None



Histograms of a few weights in 2nd junction after training for 10 epochs

contributions from Sourya Dey

Bias Initialization

Bias initialization typically does not affect performance as much as weight initialization

often the bias is initialized to zeros

may want to initialize to a small positive number
when using ReLU activations to prevent “dying"

Initializers in tf.keras

https://keras.io/initializers/

https://www.tensorflow.org/api_docs/python/tf/keras/initializers

Classes

`class Constant` : Initializer that generates tensors with constant values.

`class GlorotNormal` : The Glorot normal initializer, also called Xavier normal initializer.

`class GlorotUniform` : The Glorot uniform initializer, also called Xavier uniform initializer.

`class Identity` : Initializer that generates the identity matrix.

`class Initializer` : Initializer base class: all initializers inherit from this class.

`class Ones` : Initializer that generates tensors initialized to 1.

`class Orthogonal` : Initializer that generates an orthogonal matrix.

`class RandomNormal` : Initializer that generates tensors with a normal distribution.

`class RandomUniform` : Initializer that generates tensors with a uniform distribution.

`class TruncatedNormal` : Initializer that generates a truncated normal distribution.

`class VarianceScaling` : Initializer capable of adapting its scale to the shape of weights tensors.

`class Zeros` : Initializer that generates tensors initialized to 0.

`class constant` : Initializer that generates tensors with constant values.

`class glorot_normal` : The Glorot normal initializer, also called Xavier normal initializer.

`class glorot_uniform` : The Glorot uniform initializer, also called Xavier uniform initializer.

`class identity` : Initializer that generates the identity matrix.

`class ones` : Initializer that generates tensors initialized to 1.

`class orthogonal` : Initializer that generates an orthogonal matrix.

`class zeros` : Initializer that generates tensors initialized to 0.

Functions

`deserialize(...)` : Return an `Initializer` object from its config.

`get(...)`

`he_normal(...)` : He normal initializer.

`he_uniform(...)` : He uniform variance scaling initializer.

`lecun_normal(...)` : LeCun normal initializer.

`lecun_uniform(...)` : LeCun uniform initializer.

`serialize(...)`

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
                   activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

layers have default initializers (they work well…)

Outline for Slides

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

Cost (Loss) Functions

these are already covered, but let's review and see how they translate to tf.keras

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 M x 1 vectors

Cost (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

tf.keras implements the average (good since it is normalized for number of classes)

```
model.compile('sgd', loss=tf.keras.losses.MeanSquaredError())
```

```
ms = tf.keras.losses.MeanSquaredError()
ms([[1, 1, 1], [2,2,2]], [[0, 0, 0], [3,3,3]]).numpy().numpy()  # Loss: 1
```

for BP Initialization

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

Note: used to be mean_squared_error()

Cost (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

tf.keras implements the average (good since it is normalized for number of classes)

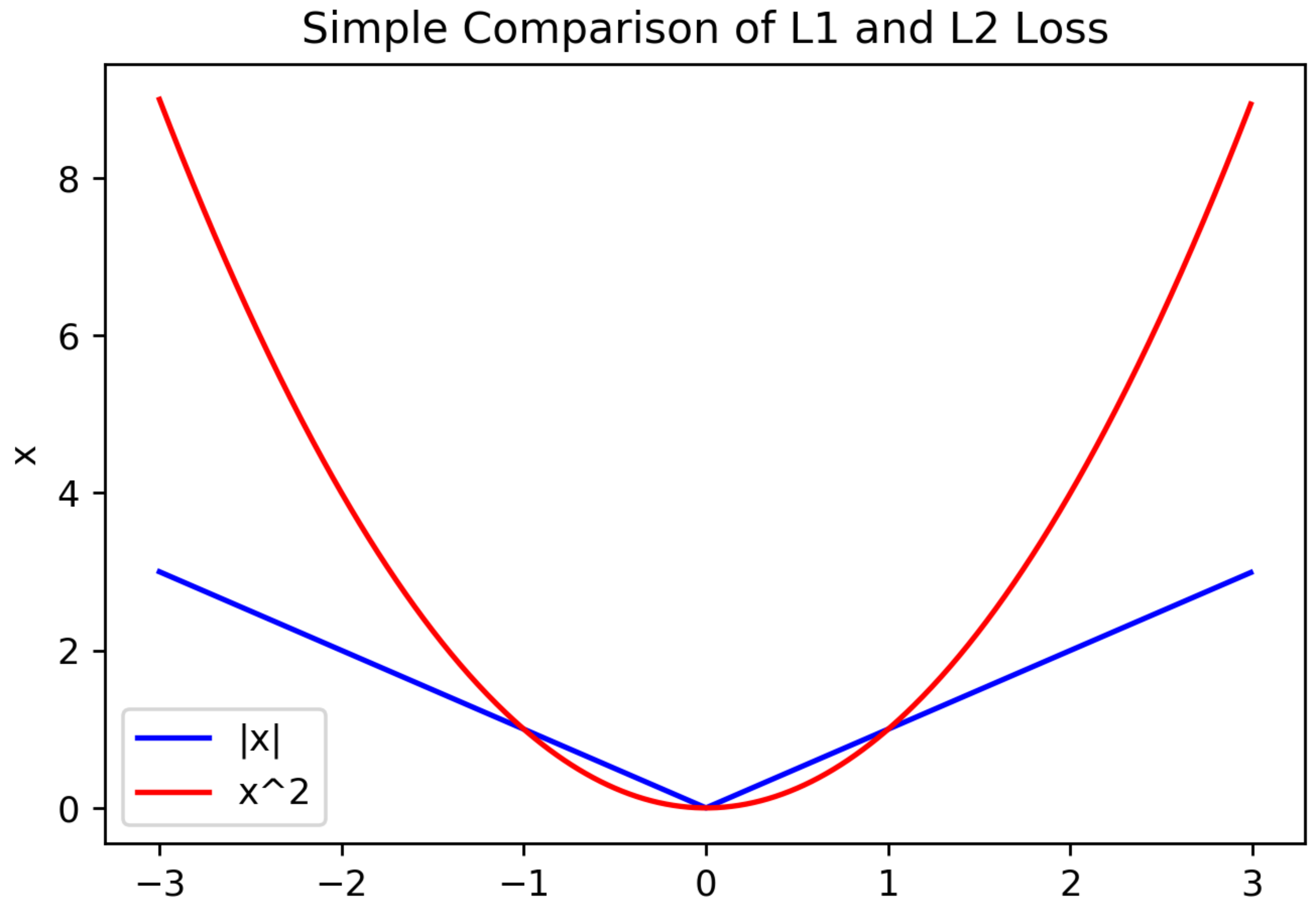
for BP Initialization

```
model.compile('sgd', loss=tf.keras.losses.MeanAbsoluteError())
```

$$\frac{d}{da}|y - a| = \mathrm{sgn}(y - a) = \begin{cases} +1 & a > y \\ -1 & a < y \end{cases}$$

Note: used to be mean_absolute_error()

Cost (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 some large weights and many other weights are near 0

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

Activations are outputs of a softmax, so interpreted as probability of class i

Cost (Loss) Functions — Multicategory Cross Entropy

Recall that with **one-hot (hard labels)** we have

$$C = -\sum_{i=1}^{M} y_i \ln a_i \longrightarrow C = -\ln a_m \quad \text{Class } m \text{ is true}$$

```
cce = tf.keras.losses.CategoricalCrossentropy()
loss = cce(
  [[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]],
  [[.9, .05, .05], [.05, .89, .06], [.05, .01, .94]])
print('Loss: ', loss.numpy())  # Loss: 0.0945
```

```
( np.log(0.9) + np.log(0.89) + np.log(0.94) ) / 3  = -0.09458991187728844
```

(averaged over batch size)

recall, in this cases, MCE is the negative-log-liklihood with regression error model:

$$p(\text{class} = i) = a_i$$

Cost (Loss) Functions — Multicategory Cross Entropy

Recall that with **soft labels** we use the general form

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

```
cce = tf.keras.losses.CategoricalCrossentropy()
loss = cce(
  [[0.7, 0.2, 0.1], [0.05, 0.9, 0.05], [0.3, 0.3, 0.4]],
  [[.9, .05, .05], [.05, .89, .06], [.05, .01, .94]])
print('Loss: ', loss.numpy())  # Loss: 1.22
```

```
y = np.asarray([[0.7, 0.2, 0.1], [0.05, 0.9, 0.05], [0.3, 0.3, 0.4]]).reshape(9)
a = np.asarray([[.9, .05, .05], [.05, .89, .06], [.05, .01, .94]]).reshape(9)
np.dot( y, -1 * np.log(a) ) / 3
1.22
```

recall, in this cases, MCE is a constant offset from the KL-divergence between the **y** and **a** probability mass functions

how do these two numerical examples compare?

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

tf.keras uses this

```
bce = tf.keras.losses.BinaryCrossentropy()
bce([[0, 1, 0]], [[0.6, 0.8, 0.1]]).numpy()  # Loss: 0.415
```

```
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]) ) )
0.41493159945336
```

Cross Entropy Loss — "From Logits"

numerically simpler (and more stable) to compute Loss(activation(s)) in one step

example: binary cross entropy

$$
\begin{aligned}
C &= -y\ln(a) - (1-a)\ln(a) \\
a &= \sigma(s) \\
&= \left[1 + e^{-s}\right]^{-1} \\
C &= y\ln(1+e^{-s}) - (1-a)\ln(1+e^{+s}) \\
&= \ln(1+e^{-\bar{y}s}) \\
\bar{y} &= (-1)^{y}
\end{aligned}
$$

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

computed 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

use "from_logits=True" in cost and linear activation on final layer

Cross Entropy Loss — "From Logits"

numerically simpler (and more stable) to compute Loss(activation(s)) in one step

example: multicategory cross entropy

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

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

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

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

computed directly from linear activation:

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

$$C = K(\mathbf{s}) - s_m \quad \text{Class } m \text{ is true, hard labels}$$

use "from_logits=True" in cost and linear activation on final layer

Cross Entropy Loss — “From Logits”

$$
\begin{aligned}
K(\mathbf{s}) &= \ln\left(\sum_j e^{s_j}\right) \\
&= \max\nolimits_j^* \; s_j \\
\max{}^*(x,y) &= \ln(e^x + e^y) \\
&= \max(x,y) + \ln\left(1 + e^{-|x-y|}\right) \\
\max{}^*(x,y,z) &= \ln(e^x + e^y + e^z) \\
&= \max{}^*\left(\max{}^*(x,y), z\right)
\end{aligned}
$$

numerically stable approach

computed directly from linear activation:

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

$$
C = \max\nolimits_j^* \; s_j - s_m \quad \text{Class } m \text{ is true, hard labels}
$$

use “from_logits=True” in cost and linear activation on final layer

Cross Entropy Loss — Variation

when your labels are Mx1 pmfs:

```
tf.keras.losses.CategoricalCrossentropy()
```

y = [0, 0, 0, 0, 0, 0, 0, 0, 1]

when your labels are hard and just the true category:

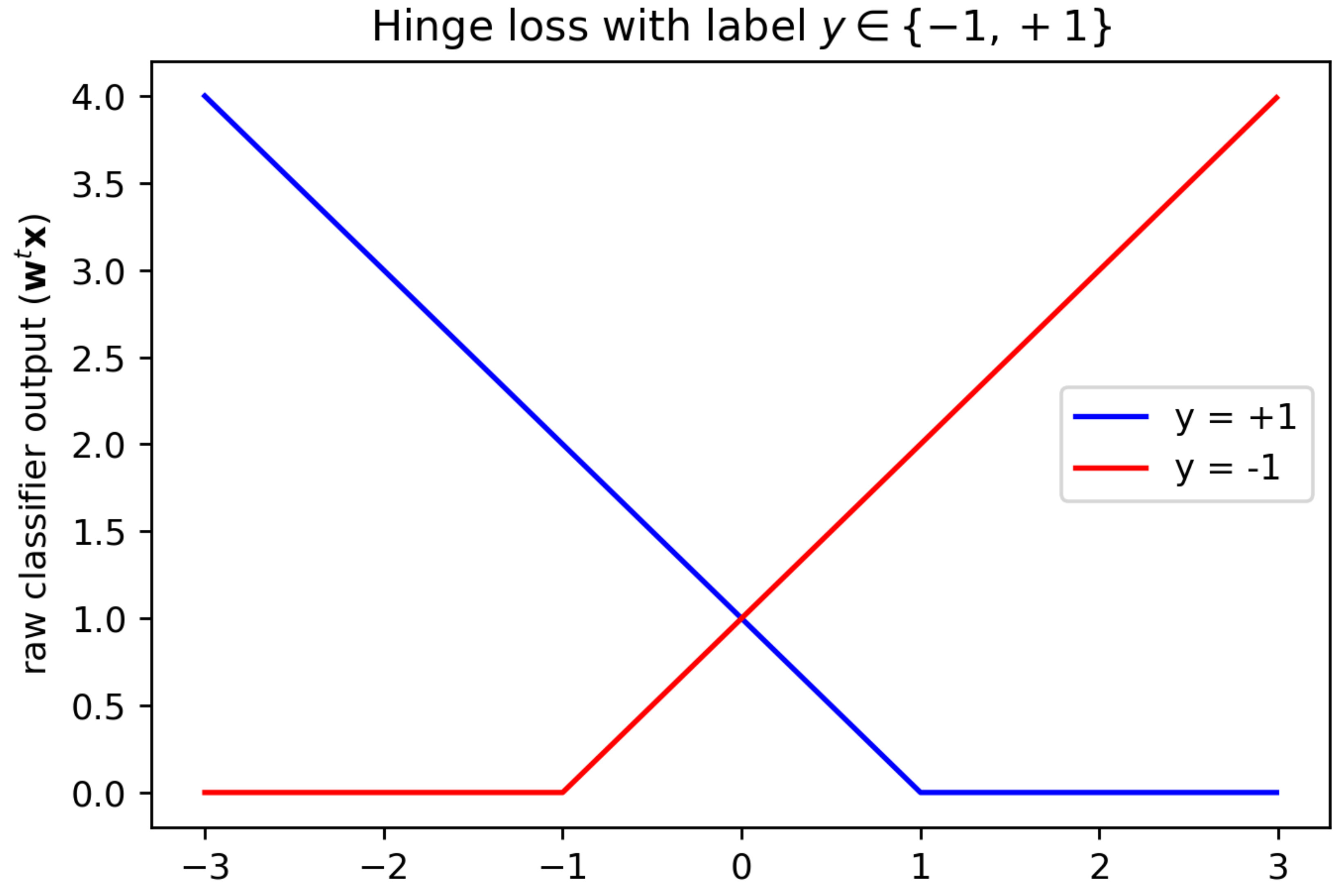
```
tf.keras.losses.SparseCategoricalCrossentropy()
```

y = 9

y = 9 $\xrightarrow{\texttt{tf.keras.utils.to\_categorical()}}$ y = [0, 0, 0, 0, 0, 0, 0, 0, 1]

Hinge Loss

for binary classifier (target/labels in {-1,+1})



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

typically use linear output activation

```
model.compile('sgd', loss=tf.keras.losses.Hinge())
```

Loss Function in tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/losses

Classes

`class BinaryCrossentropy` : Computes the cross-entropy loss between true labels and predicted labels.

`class CategoricalCrossentropy` : Computes the crossentropy loss between the labels and predictions.

`class CategoricalHinge` : Computes the categorical hinge loss between `y_true` and `y_pred`.

`class CosineSimilarity` : Computes the cosine similarity between `y_true` and `y_pred`.

`class Hinge` : Computes the hinge loss between `y_true` and `y_pred`.

`class Huber` : Computes the Huber loss between `y_true` and `y_pred`.

`class KLDivergence` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred`.

`class LogCosh` : Computes the logarithm of the hyperbolic cosine of the prediction error.

`class Loss` : Loss base class.

`class MeanAbsoluteError` : Computes the mean of absolute difference between labels and predictions.

`class MeanAbsolutePercentageError` : Computes the mean absolute percentage error between `y_true` and `y_pred`.

`class MeanSquaredError` : Computes the mean of squares of errors between labels and predictions.

`class MeanSquaredLogarithmicError` : Computes the mean squared logarithmic error between `y_true` and `y_pred`.

`class Poisson` : Computes the Poisson loss between `y_true` and `y_pred`.

`class Reduction` : Types of loss reduction.

`class SparseCategoricalCrossentropy` : Computes the crossentropy loss between the labels and predictions.

`class SquaredHinge` : Computes the squared hinge loss between `y_true` and `y_pred`.

Functions

`KLD(...)` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred`.

`MAE(...)`

`MAPE(...)`

`MSE(...)`

`MSLE(...)`

`binary_crossentropy(...)`

`categorical_crossentropy(...)` : Computes the categorical crossentropy loss.

`categorical_hinge(...)` : Computes the categorical hinge loss between `y_true` and `y_pred`.

`cosine_similarity(...)` : Computes the cosine similarity between labels and predictions.

`deserialize(...)`

`get(...)`

`hinge(...)` : Computes the hinge loss between `y_true` and `y_pred`.

`kld(...)` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred`.

`kullback_leibler_divergence(...)` : Computes Kullback-Leibler divergence loss between `y_true` and `y_pred`.

`logcosh(...)` : Logarithm of the hyperbolic cosine of the prediction error.

`mae(...)`

`mape(...)`

`mean_absolute_error(...)`

`mean_absolute_percentage_error(...)`

`mean_squared_error(...)`

`mean_squared_logarithmic_error(...)`

`mse(...)`

`msle(...)`

`poisson(...)` : Computes the Poisson loss between y_true and y_pred.

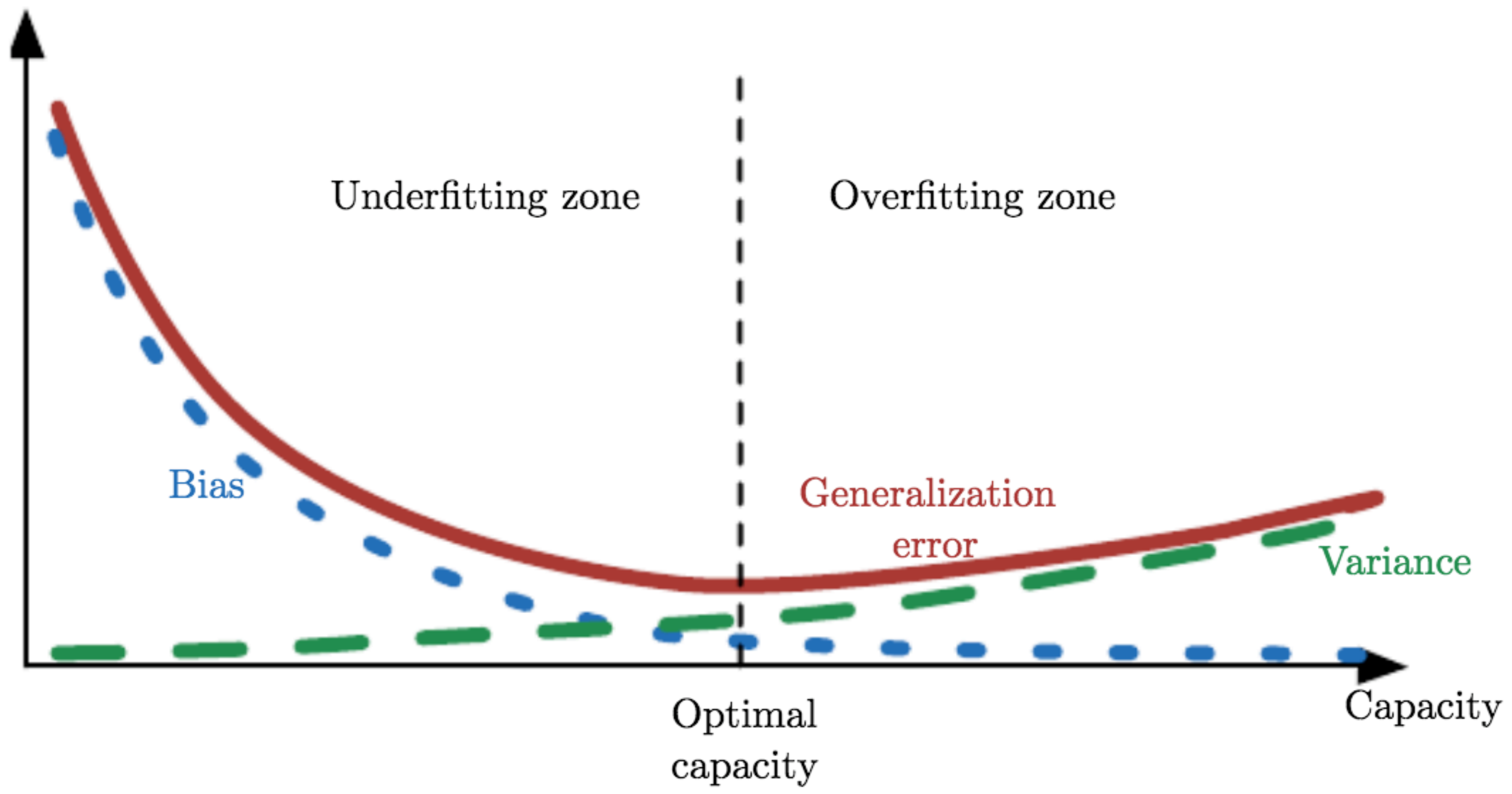
`serialize(...)`

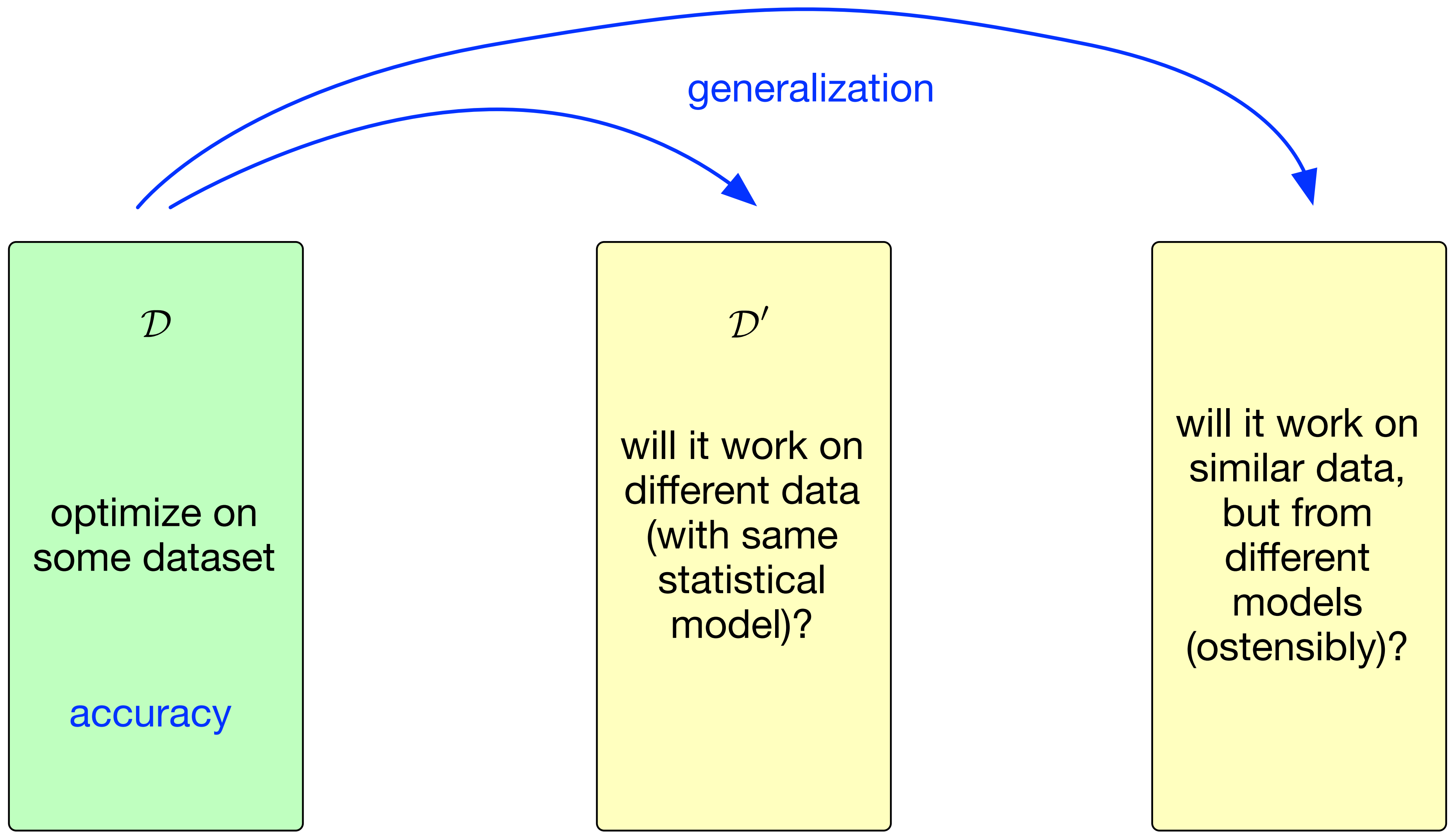
`sparse_categorical_crossentropy(...)`

`squared_hinge(...)` : Computes the squared hinge loss between `y_true` and `y_pred`.

Default loss is None, so you need to specify the loss to run model.compile()

Regularizers — Why?





the trade-off between over and under fitting is often called 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 — What?

Main goal of Machine Learning is to
GENERALIZE

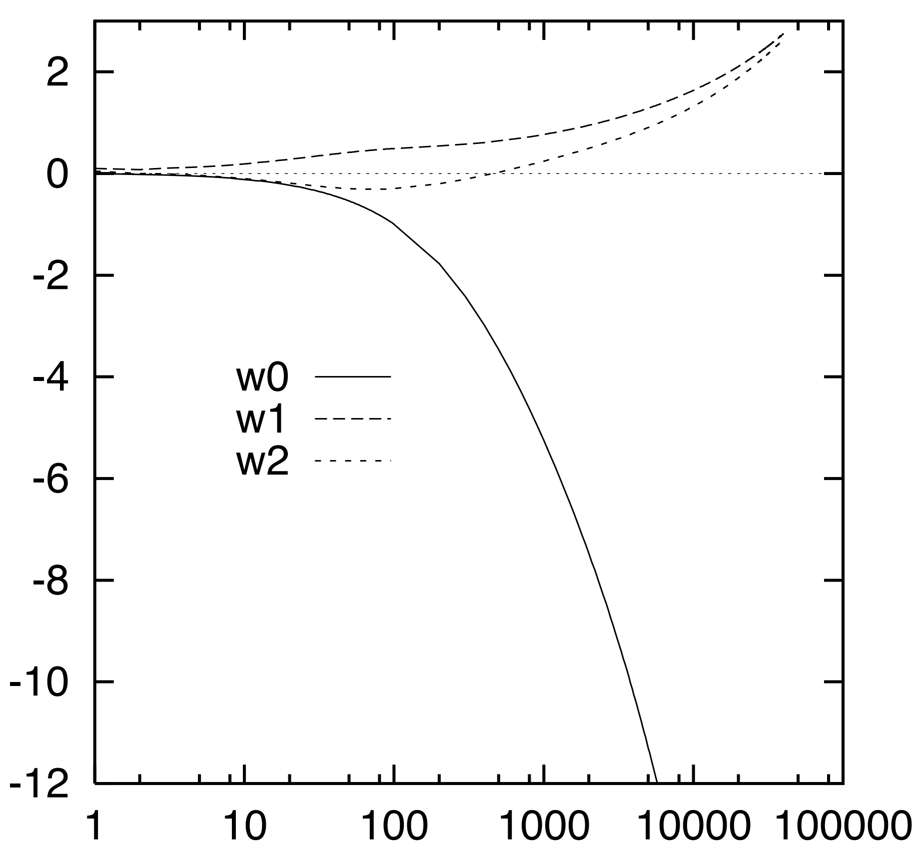
**regularization is anything you do in training that is aimed at improving generalization over accuracy —
i.e., anything that does not optimize the cost on the training data**

When people say "regularizer" they usually are using a narrower definition:

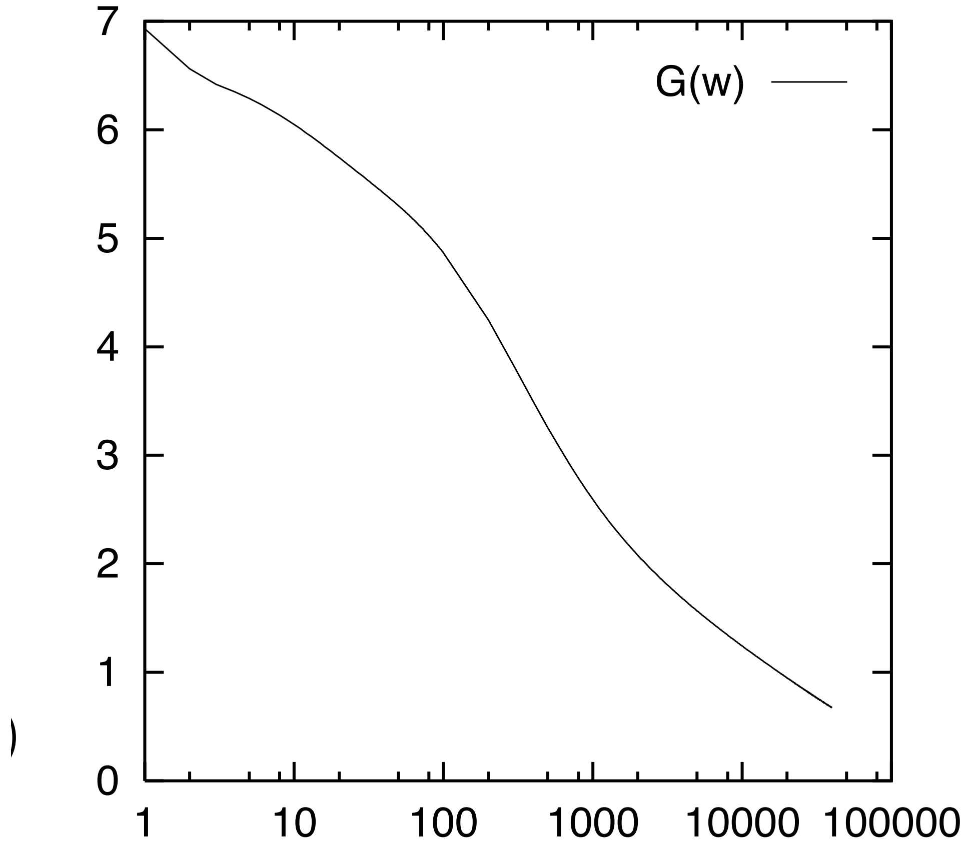
an additive term to the loss function that prevents weights from getting too large

Regularizers — How?

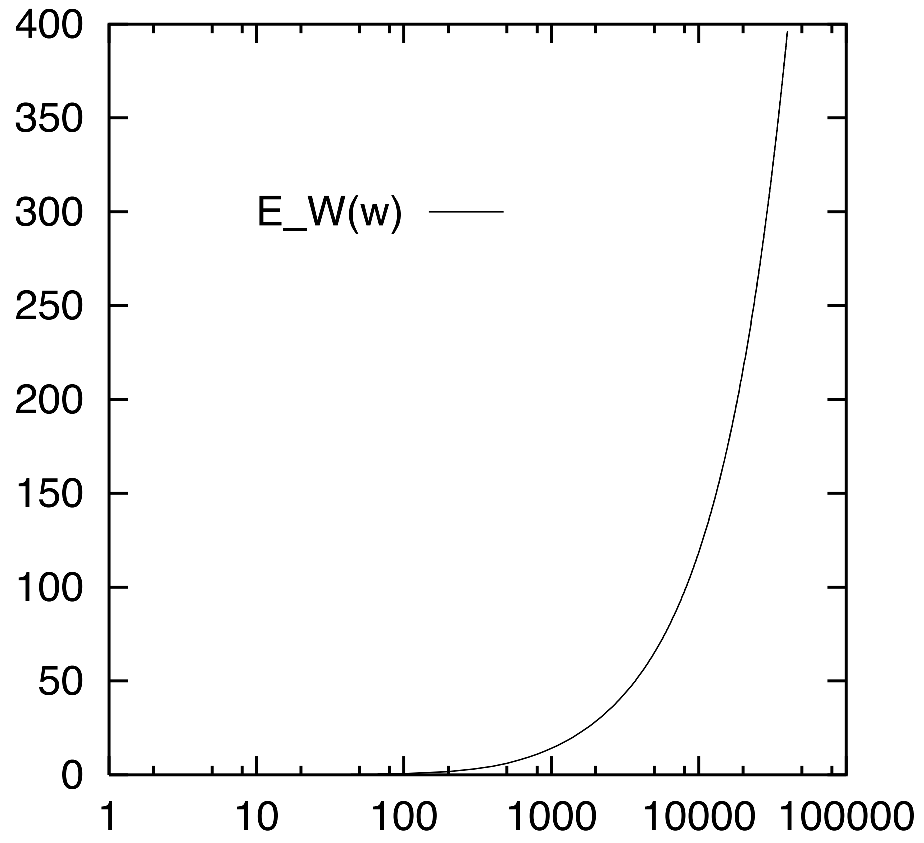
Why do large weights correspond to over-fitting???



weight evolution



learning curve (loss)

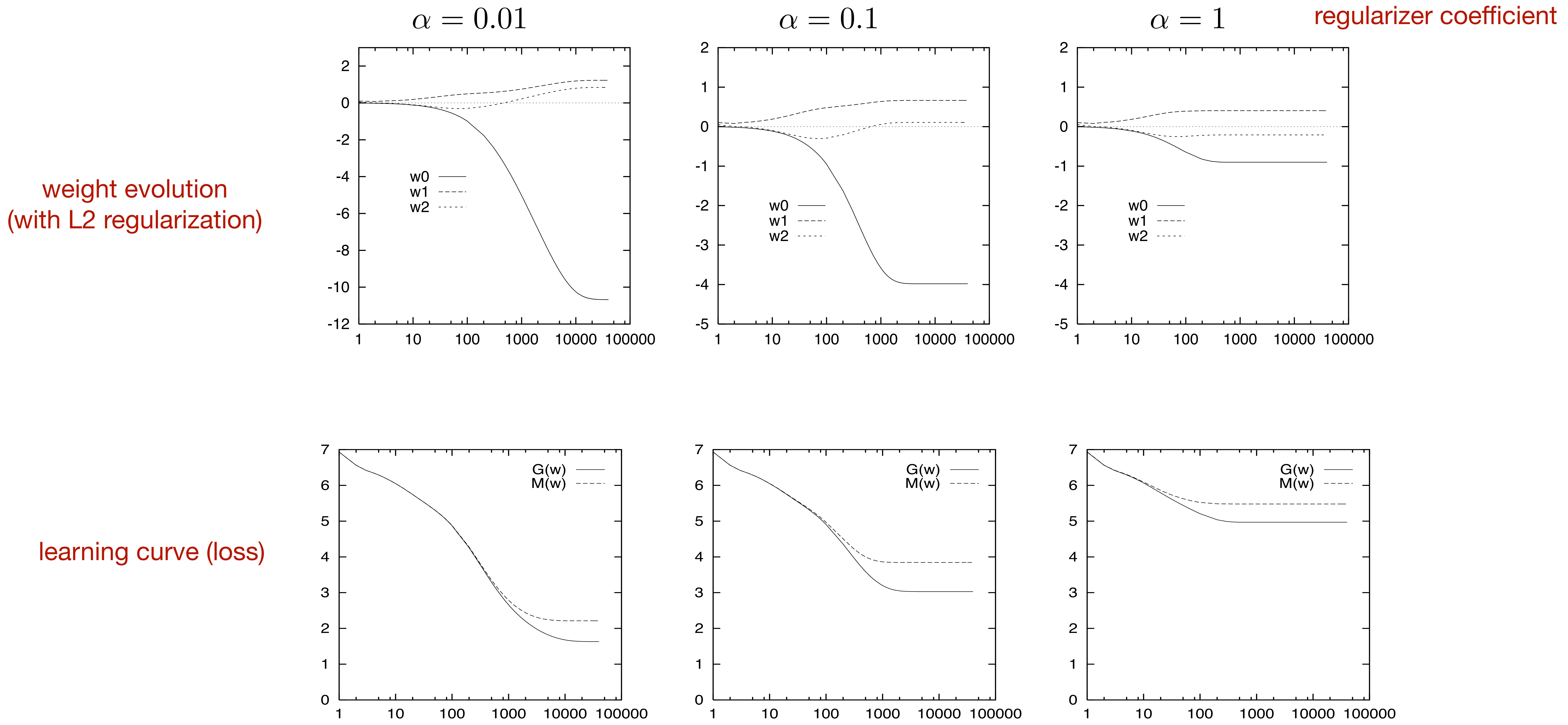


L2 norm of weights

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

Regularizers — How?

This is an experimental observation



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

Regularizers — L1, L2

L2 regularization
(aka weight decay)

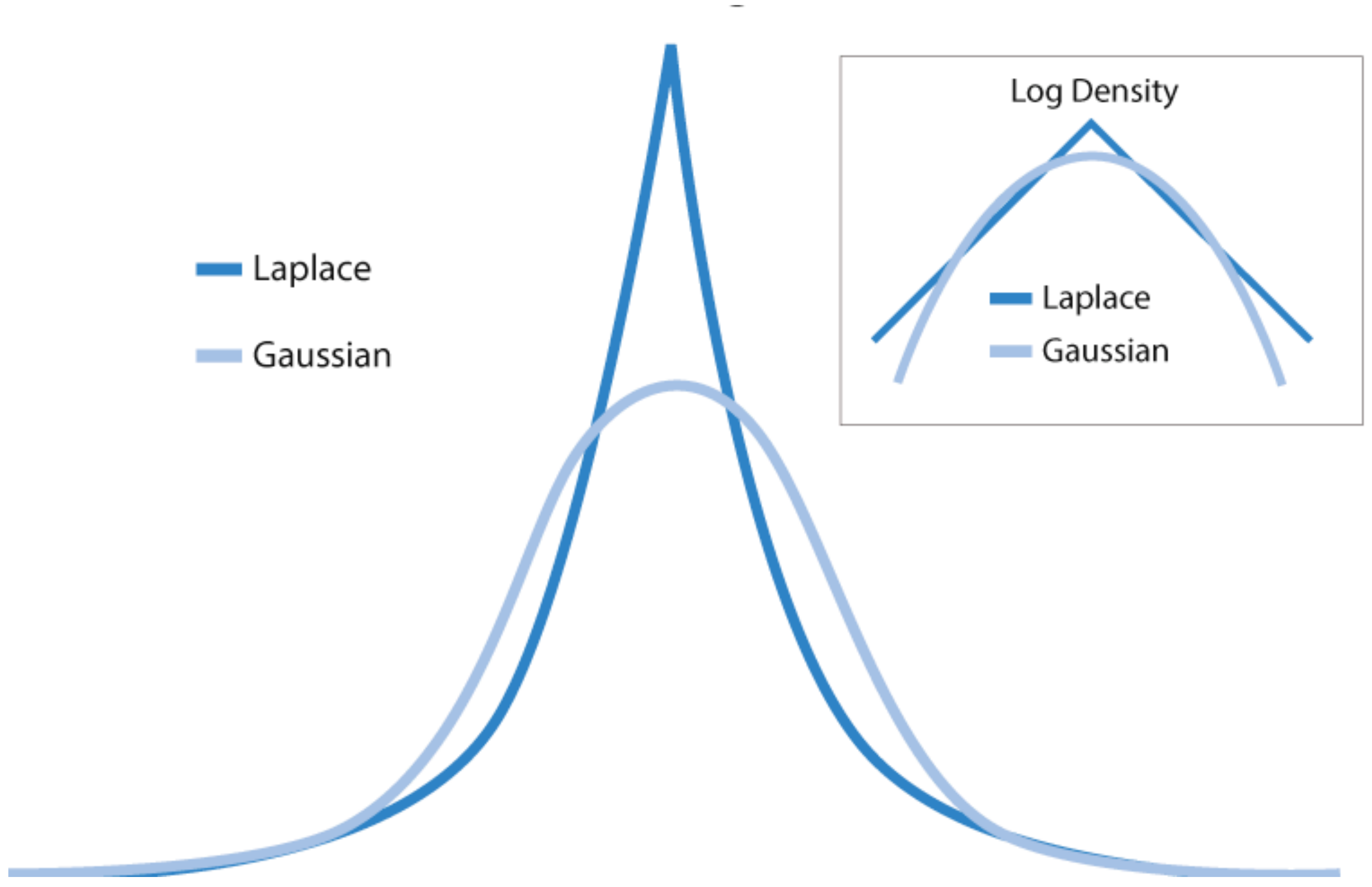
$$C = C_{\text{no-reg}} + \lambda \|\mathbf{w}\|_2^2 \quad \longrightarrow \quad w \leftarrow w - \eta \left(\frac{\partial C}{\partial w} + 2\lambda w \right)$$

L1 regularization
(aka LASSO)

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

As seen 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



contributions from Sourya Dey

Regularizers

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

$\lambda = 0$ $\longrightarrow$ $\mathbf{w}^* \sim \arg\min C_{\text{no-reg}}(\mathbf{w})$

could be **over-fitting**, depends on capacity of model, dataset properties, and inference problem

$\lambda \to \infty$ $\longrightarrow$ $\mathbf{w}^* \sim \mathbf{0}$

under-fitting

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

contributions from Sourya Dey

Regularizers in tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/regularizers?hl=de

Classes

`class L1L2` : A regularizer that applies both L1 and L2 regularization penalties.

`class Regularizer` : Regularizer base class.

Functions

`deserialize(...)`

`get(...)`

`l1(...)` : Create a regularizer that applies an L1 regularization penalty.

`l1_l2(...)` : Create a regularizer that applies both L1 and L2 penalties.

`l2(...)` : Create a regularizer that applies an L2 regularization penalty.

`serialize(...)`

default regularizer is None

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None,
                   activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
```

Let's Try Regularization Out…

```
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.utils import plot_model
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.models import load_model
from tensorflow.keras import regularizers

```

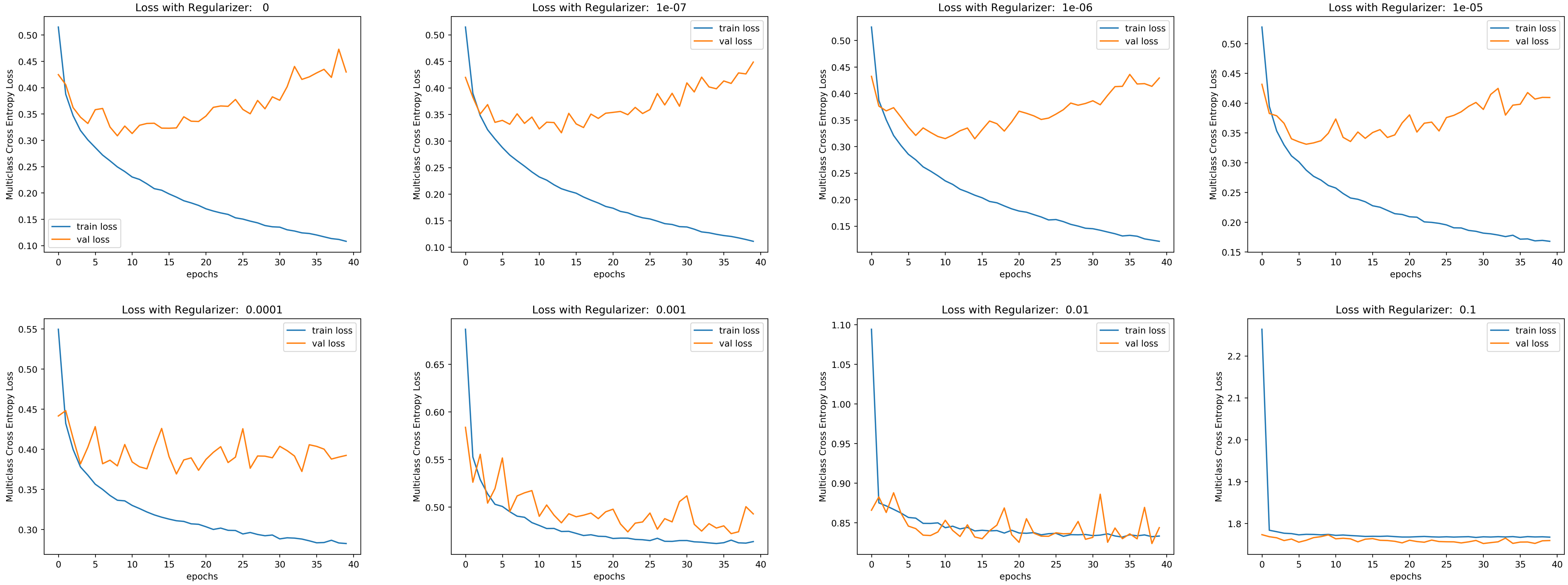
4_fmnist.py

demonstrate a different import pattern….

```
# this uses the Functional API for definning the model
reg_val = 1e-5
nnet_inputs = Input(shape=(num_pixels,), name='images')
z = Dense(128, activation='relu', kernel_regularizer=regularizers.l2(reg_val), bias_regularizer=regularizers.l2(reg_val), name='hidden')(nnet_inputs)
z = Dense(10, activation='softmax', kernel_regularizer=regularizers.l2(reg_val), bias_regularizer=regularizers.l2(reg_val), name='output')(z)

```

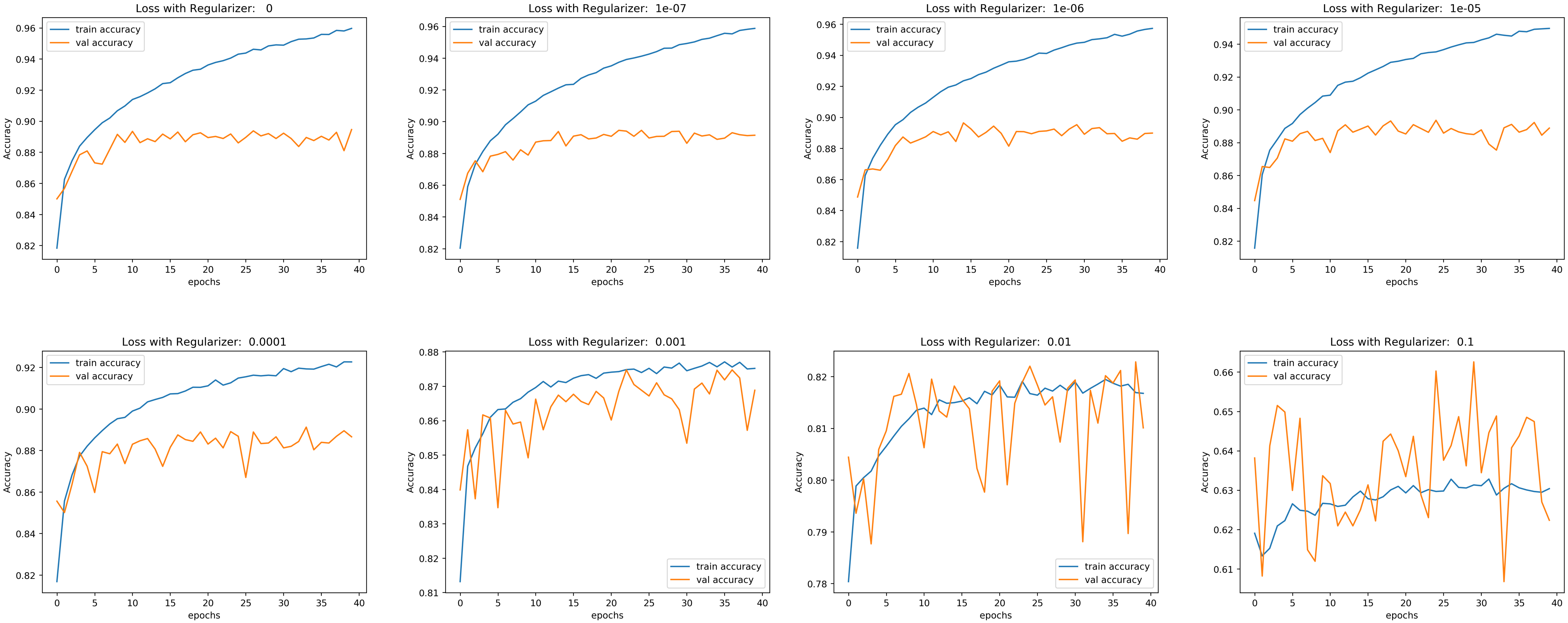
added a L2 regularizer to both layers -- used same regularizer coefficient for all weights and biases

Let’s Try L2 Regularization Out…



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

Let's Try L2 Regularization Out…



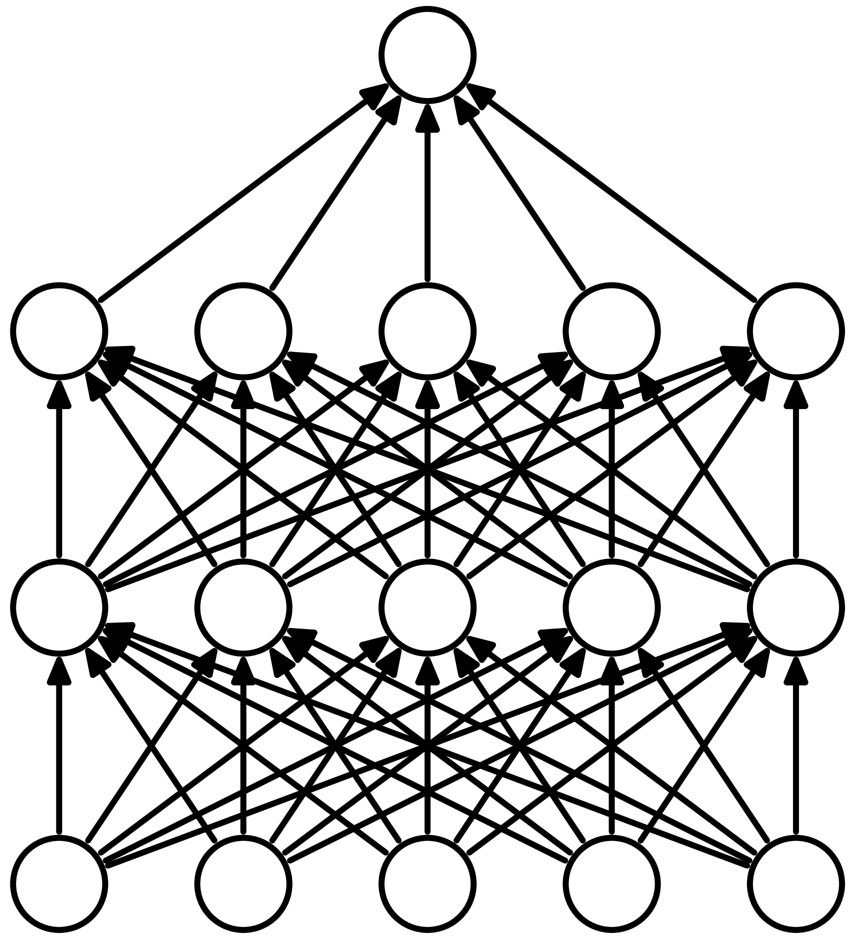
same trend as the loss…
(note: this is with 80/20 train/loss split)

this is not totally satisfying!

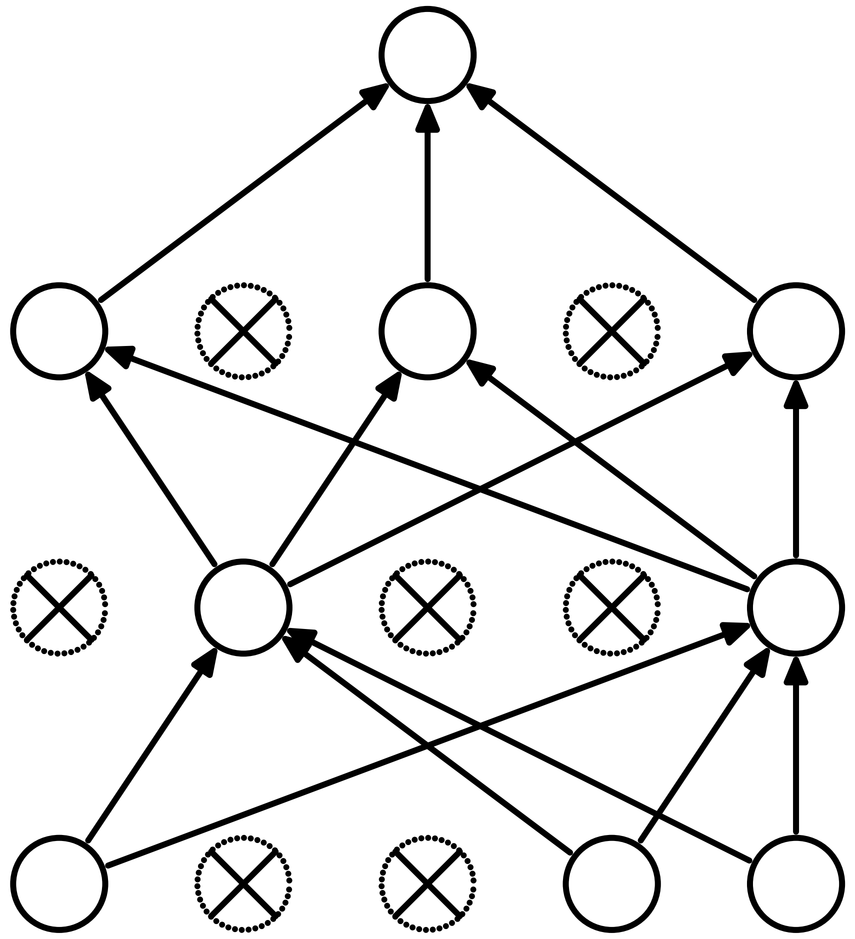
Dropout — A Different Type of Regularization

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

contributions from Sourya Dey

Dropout

very effective at reducing over fitting and improving generalization

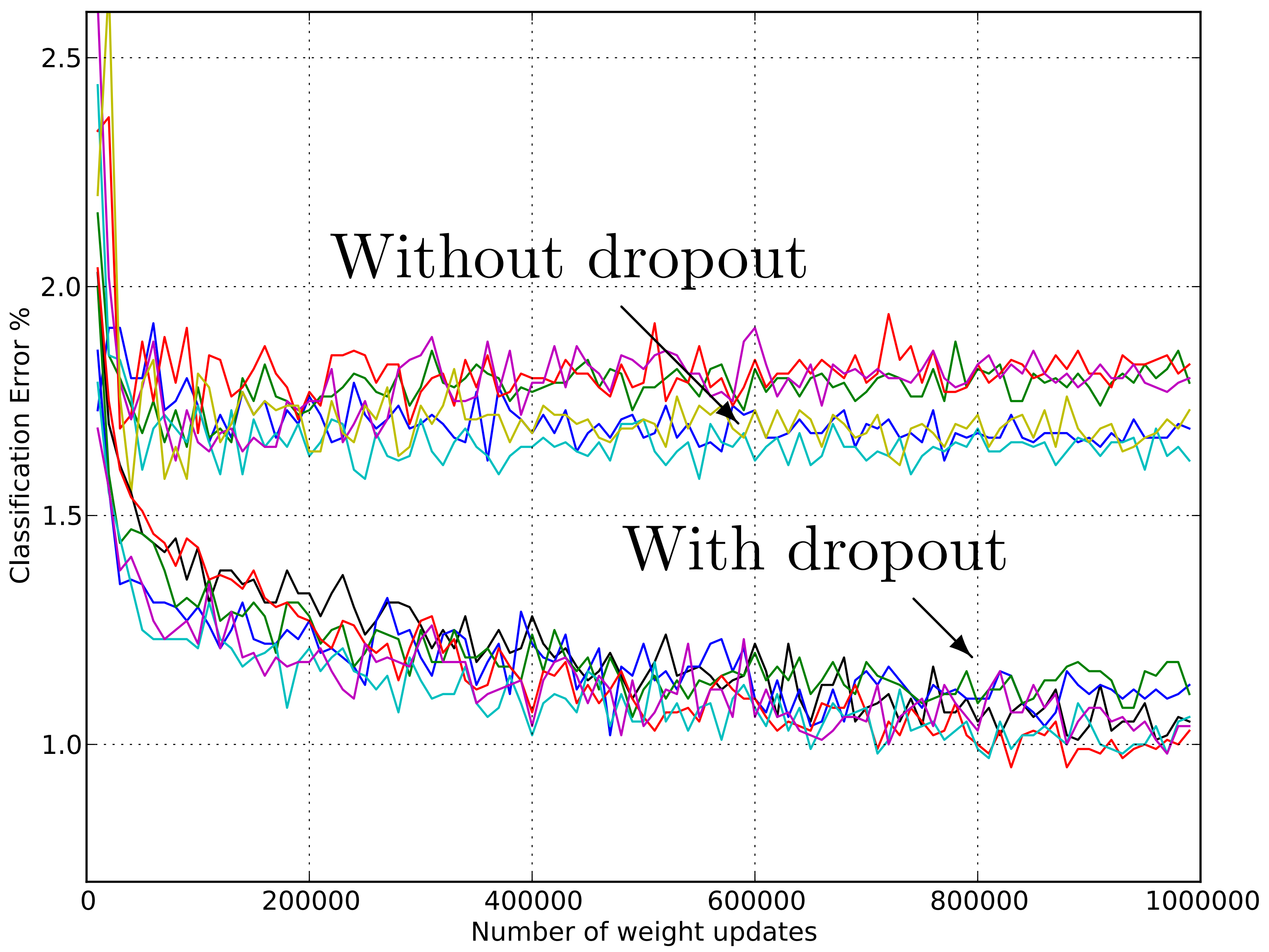
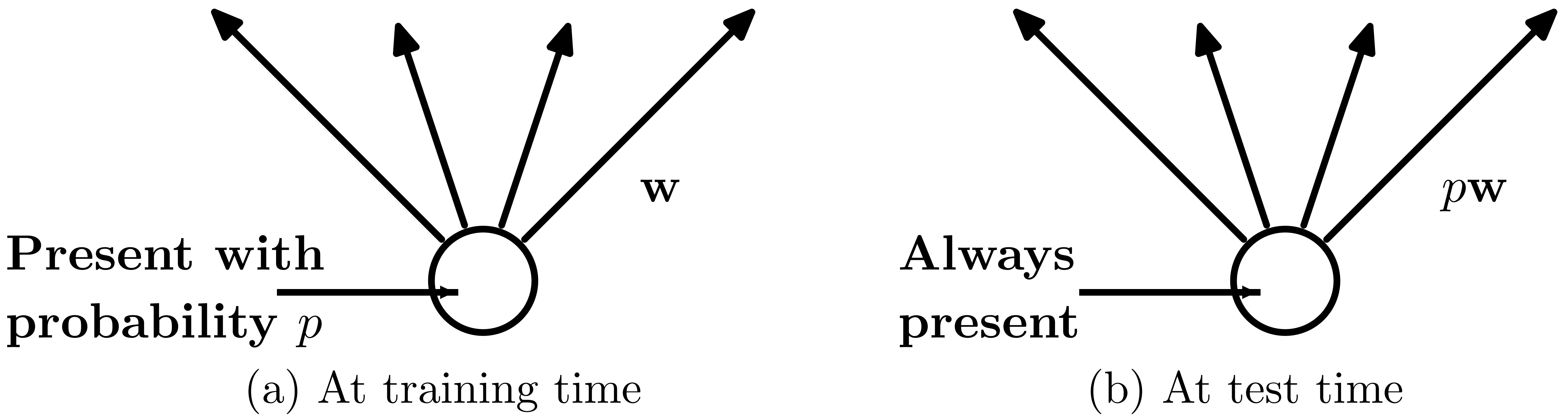


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

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 the nodes are present



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

(sort of ad hoc because of nonlinearities, but this works!)

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

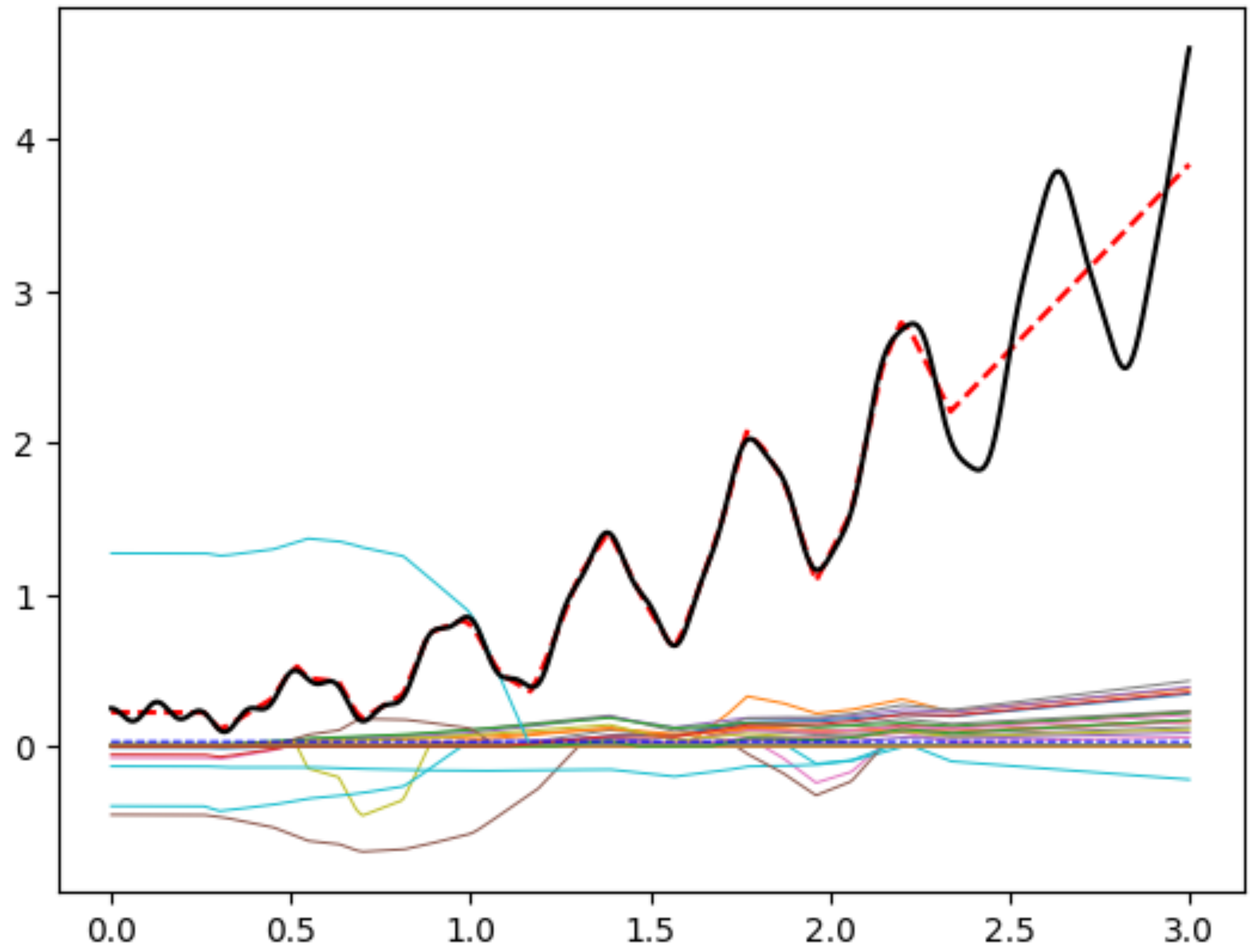
Dropout Example

http://neuralnetworksanddeeplearning.com/chap4.html

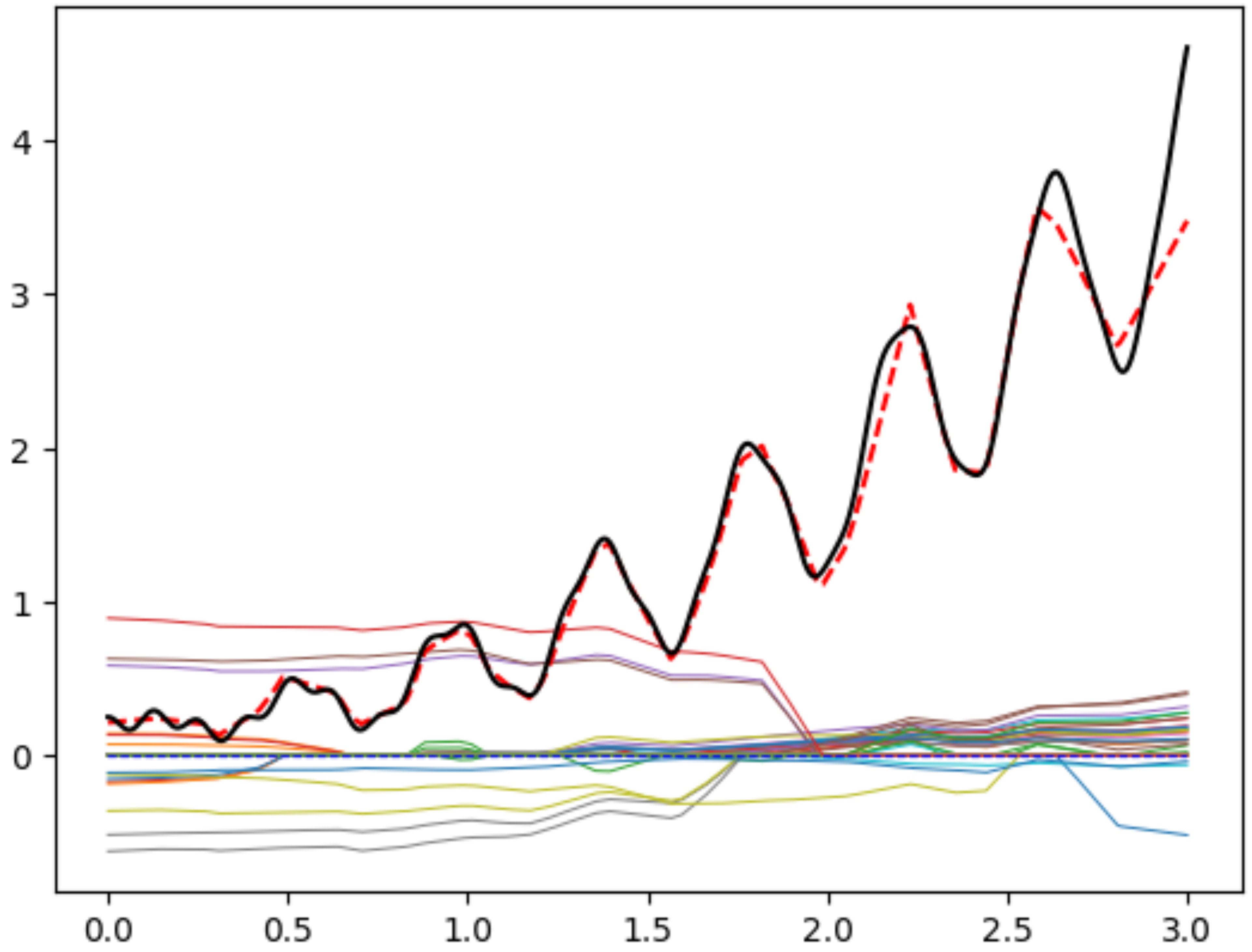
What happens when we train a neural net on Neilson's crazy function?

```
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



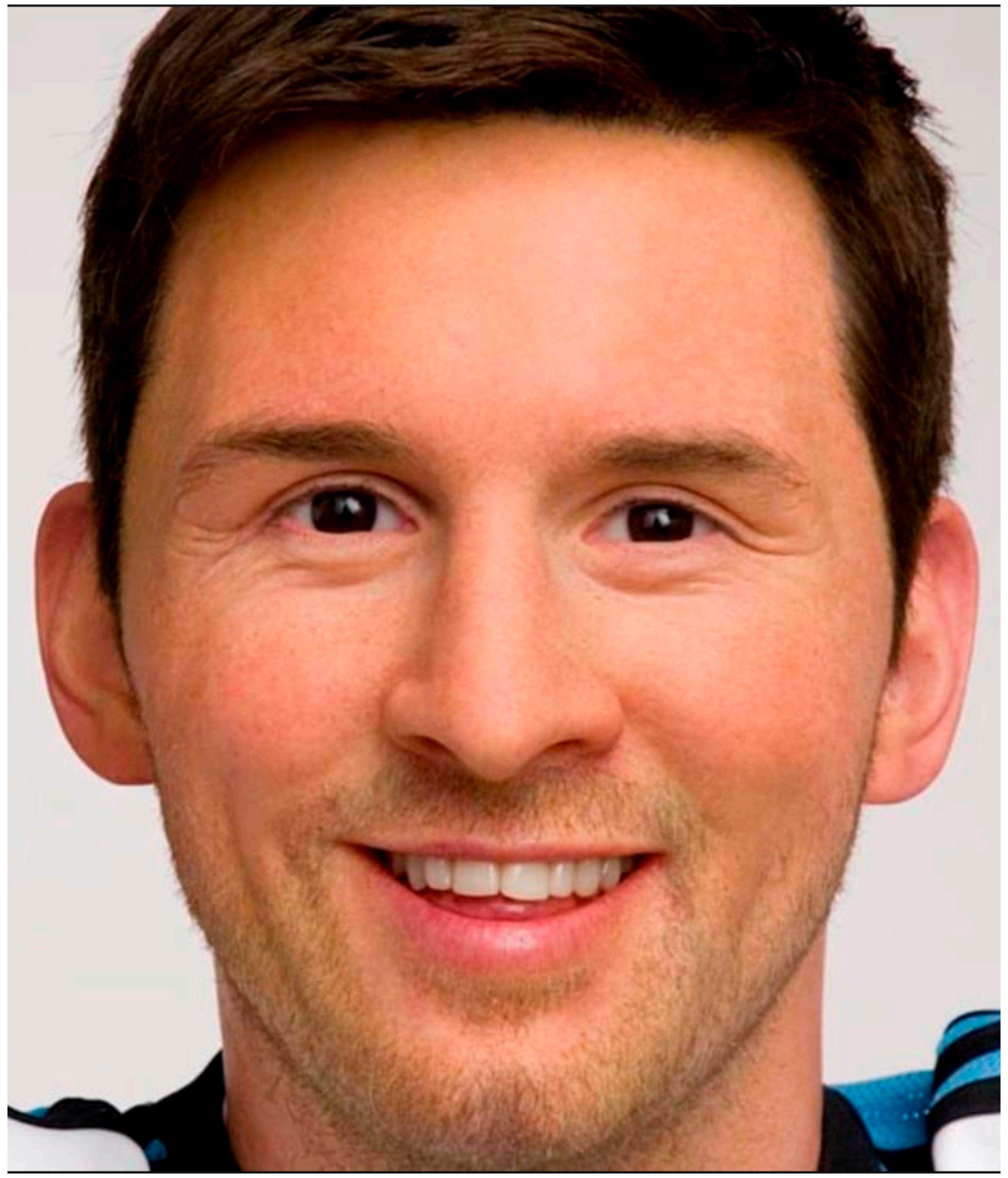
20% Dropout

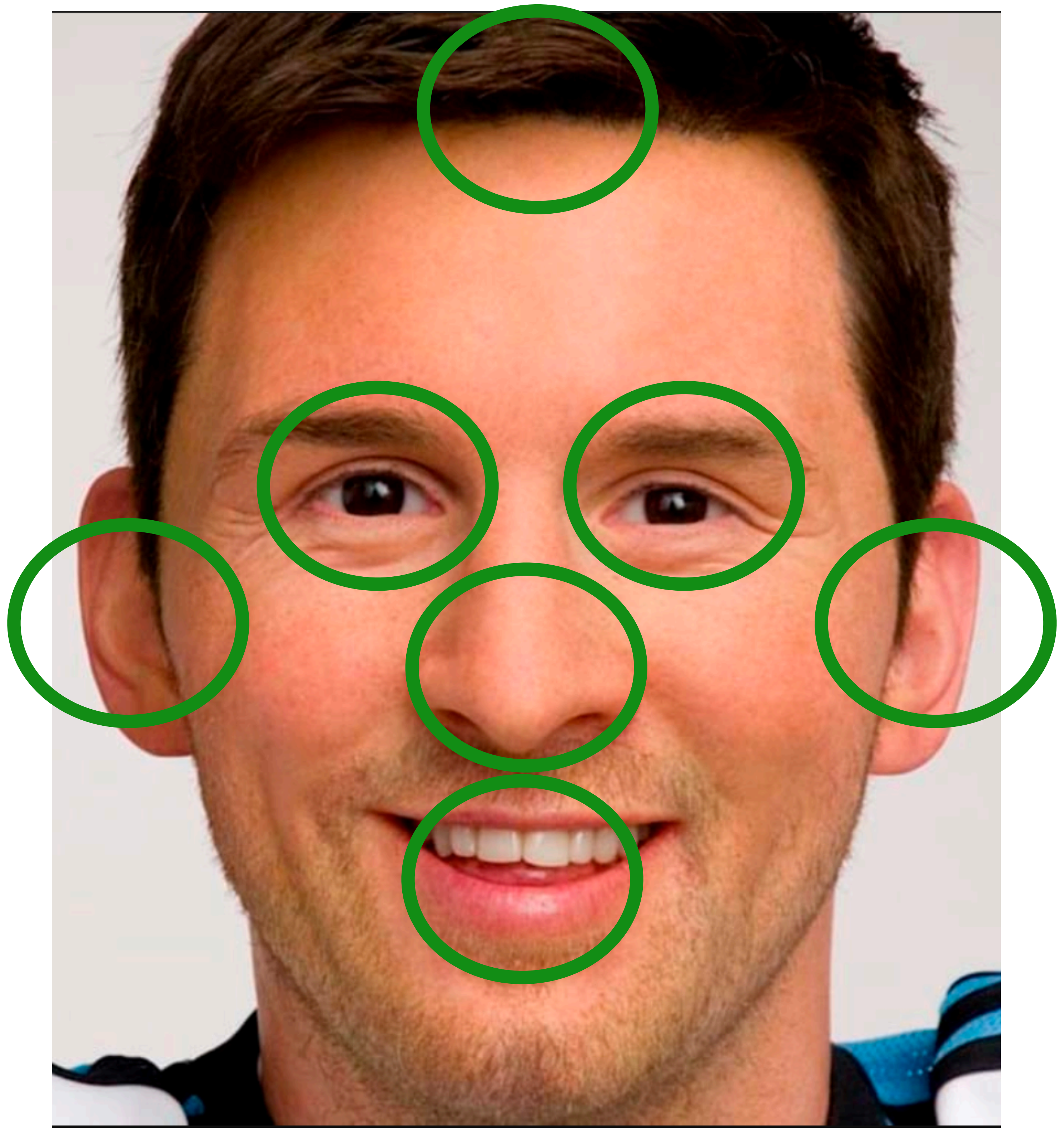
Dropout Intuition

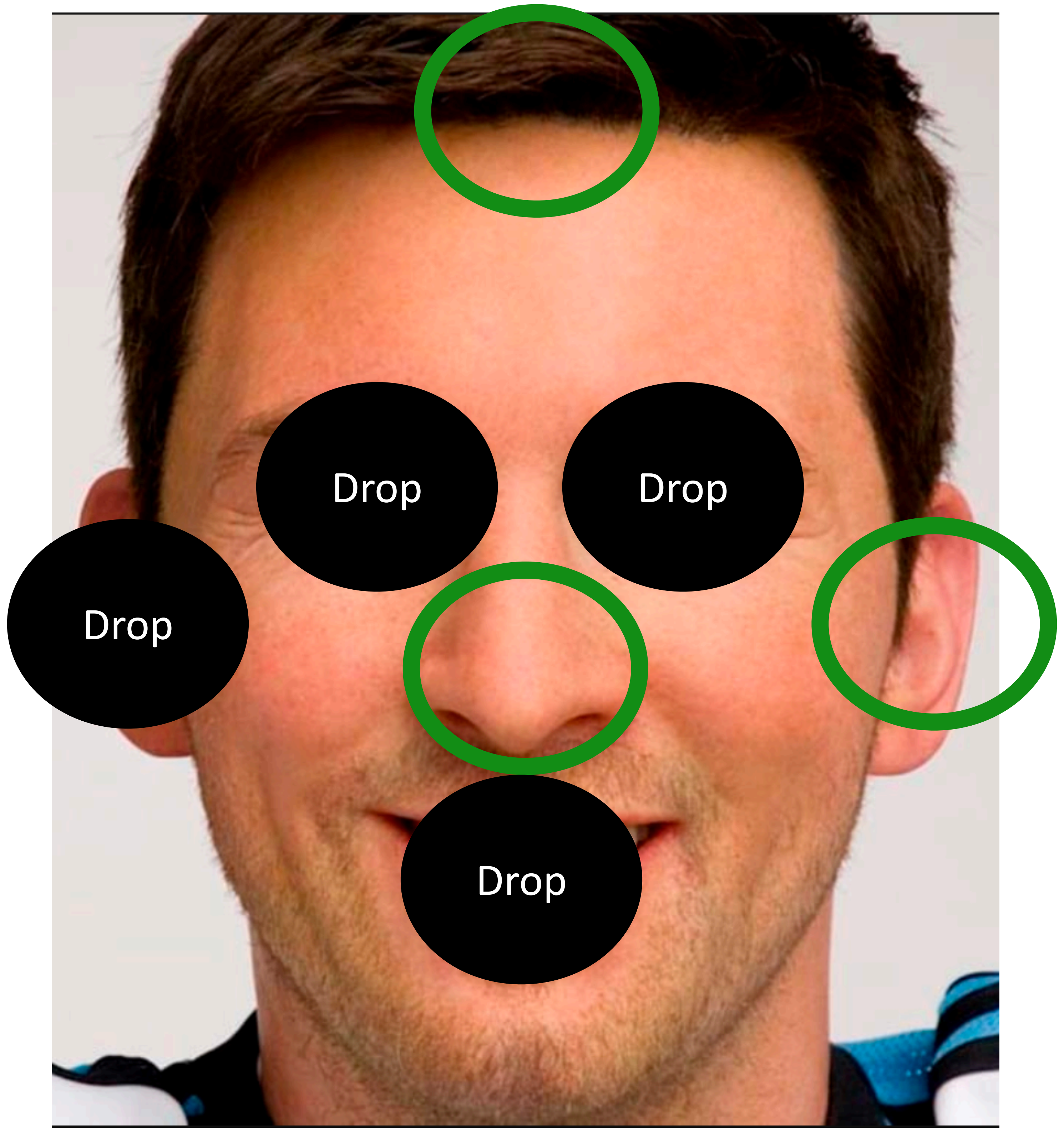
Ensemble methods: train multiple networks for same task and average

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

individual (or small groups of) nodes have to be able to do a reasonable job on the task w/o the deleted nodes ==>
Robustness/Generalization





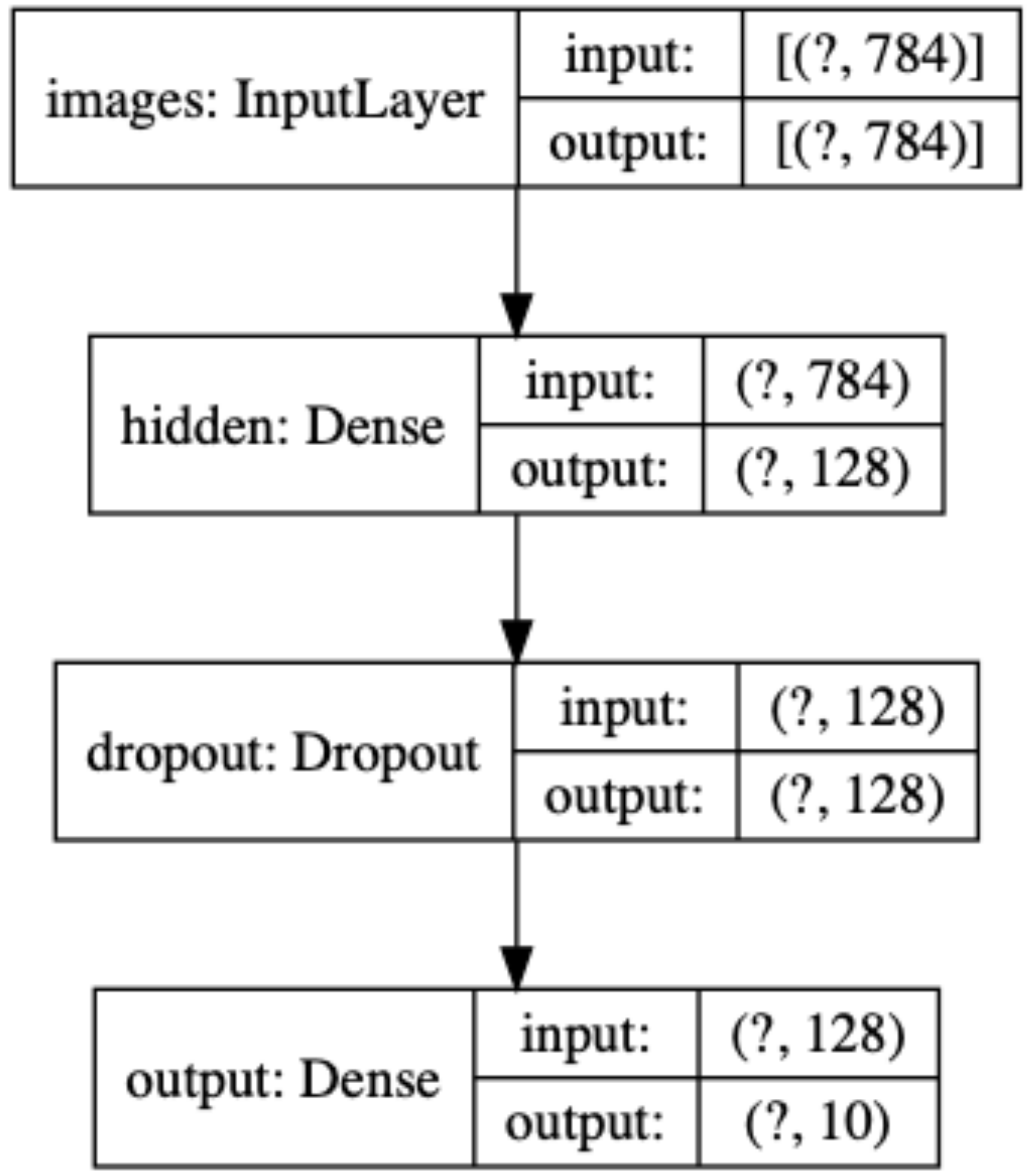


contributions from Sourya Dey

Dropout in tf.keras

```python
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.utils import plot_model
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.models import load_model
from tensorflow.keras import regularizers


import numpy as np
import matplotlib as mpl
mpl.use('Agg')  # this is to set the matplotlib backend, you may not need
import matplotlib.pyplot as plt

## these could be read with an arg-parser
reg_val = 0
dropout_rate = 0.15

### such a small run, we can hide any GPUs and run on the CPU.
### for small jobs, it can be faster on a CPU (true in this case)
import os
os.environ['CUDA_DEVICE_ORDER']='PCI_BUS_ID'
os.environ['CUDA_VISIBLE_DEVICES']=''

#### get the daatset
(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
# train_images.shape is (60000, 28, 28)
#test_images.shape (10000, 28, 28)
num_pixels = 28 * 28
train_images = train_images.reshape( (60000, num_pixels) ).astype(np.float32) / 255.0
test_images = test_images.reshape( (10000, num_pixels) ).astype(np.float32)  / 255.0

# this uses the Functional API for definning the model
nnet_inputs = Input(shape=(num_pixels,), name='images')
z = Dense(128, activation='relu', kernel_regularizer=regularizers.l2(reg_val), bias_regularizer=regularizers.l2(reg_val), name='hidden')(nnet_inputs)
z = Dropout(dropout_rate)(z)
z = Dense(10, activation='softmax', kernel_regularizer=regularizers.l2(reg_val), bias_regularizer=regularizers.l2(reg_val), name='output')(z)
```

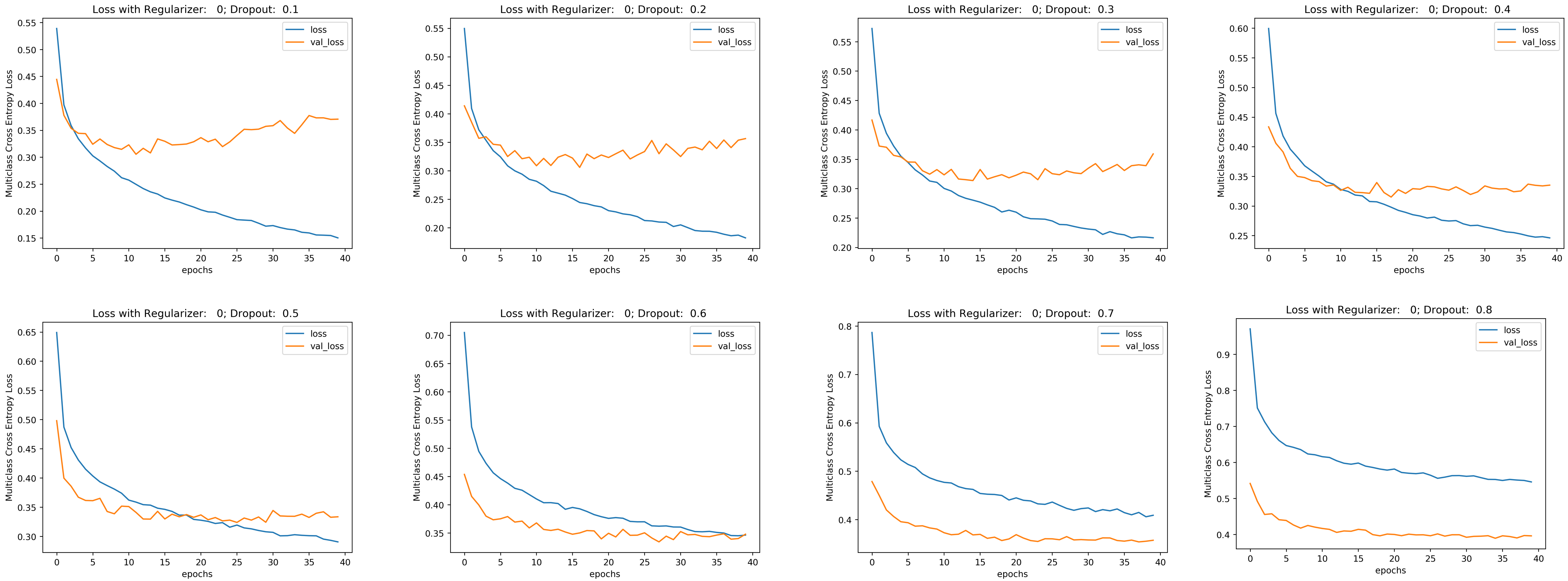
```
Layer (type)                 Output Shape              Param #
=================================================================
images (InputLayer)          [(None, 784)]             0
_________________________________________________________________
hidden (Dense)               (None, 128)               100480
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
output (Dense)               (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
```



Dropout layer has no trainable parameters — think of it as just the on/off mask that follows each node in the Dense layer

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

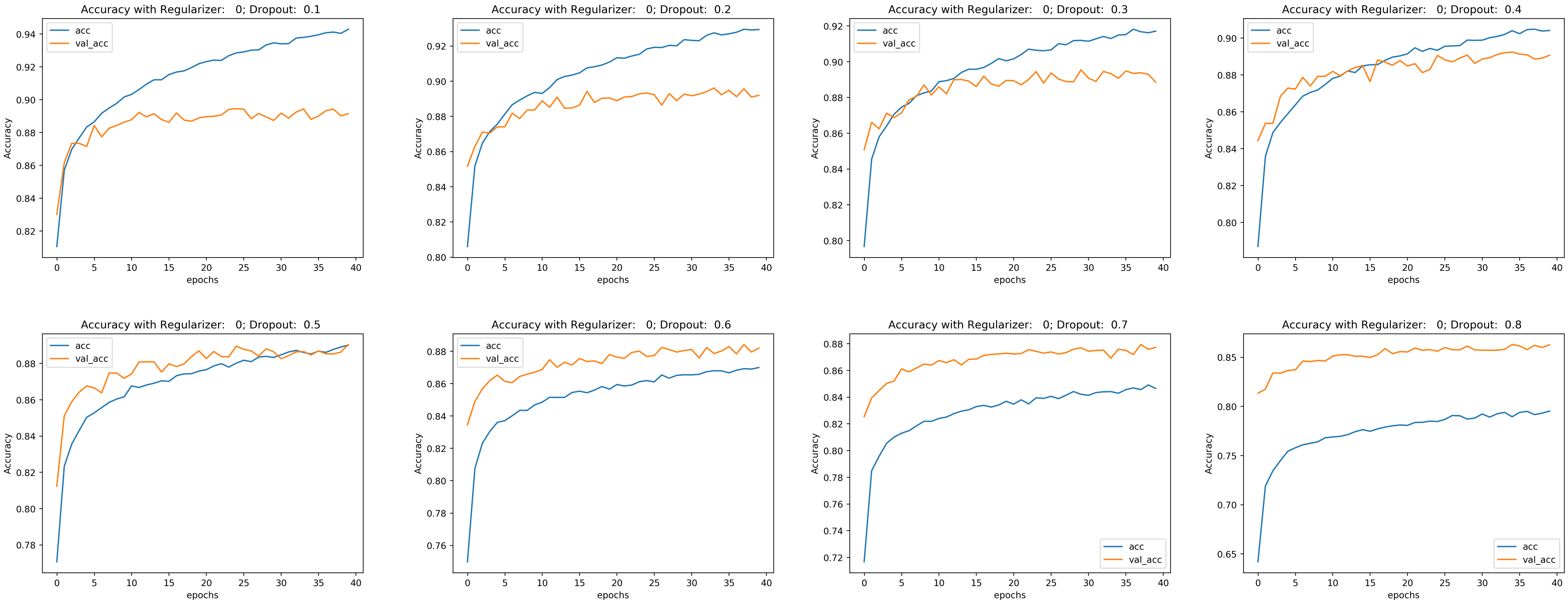
Dropout with no L2 Regularization



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

(better than L2 regularization in this case)

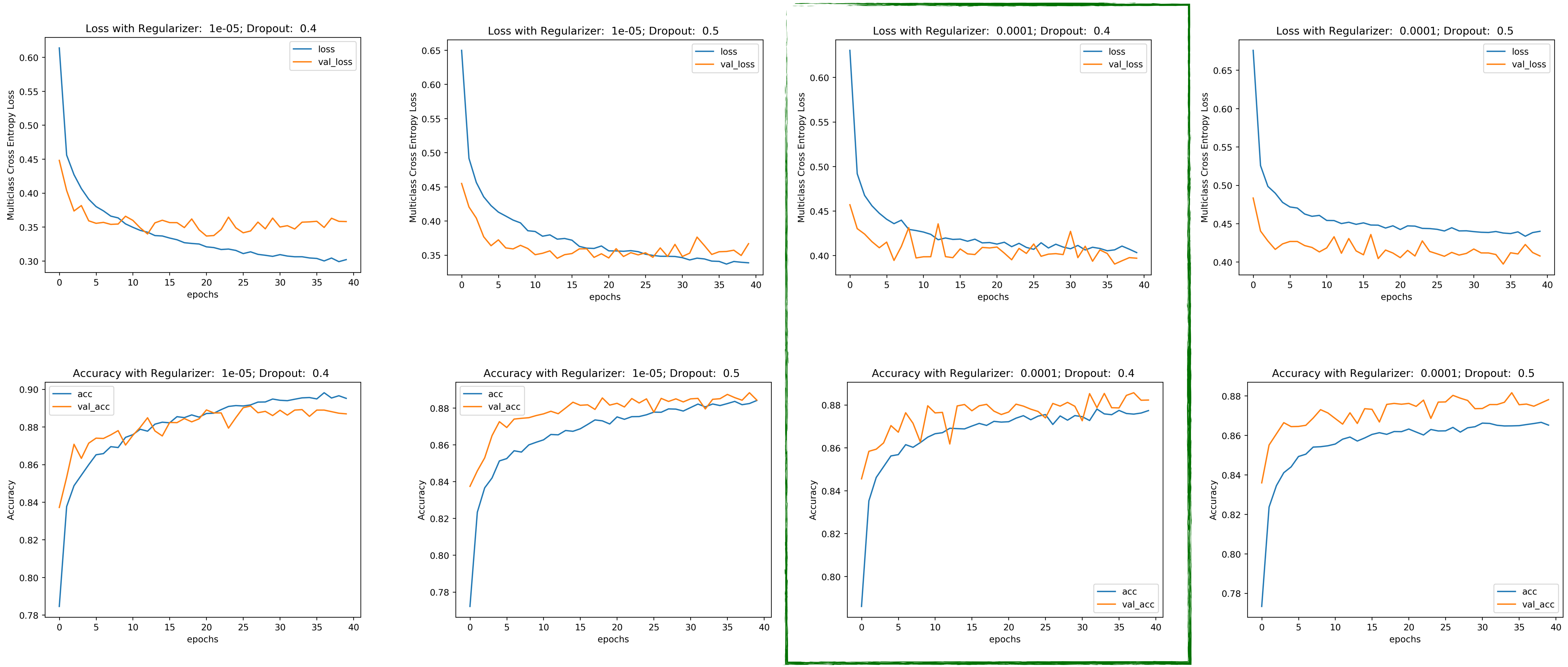
Dropout with no L2 Regularization



similar trend as loss

(better than L2 regularization in this case)

Dropout and L2 Regularization



this achieves a 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…

What does this suggest to you??

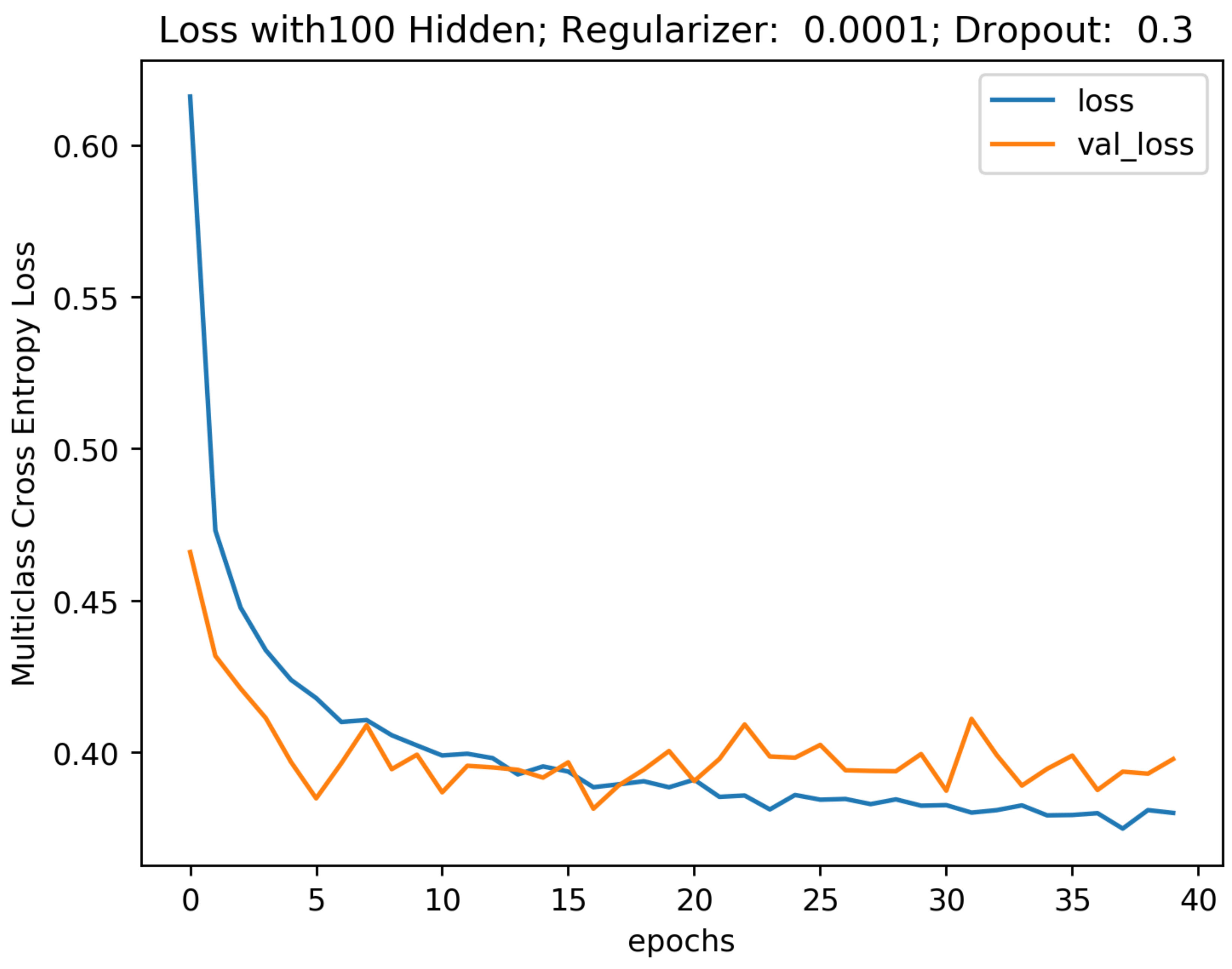
Nominal Values:

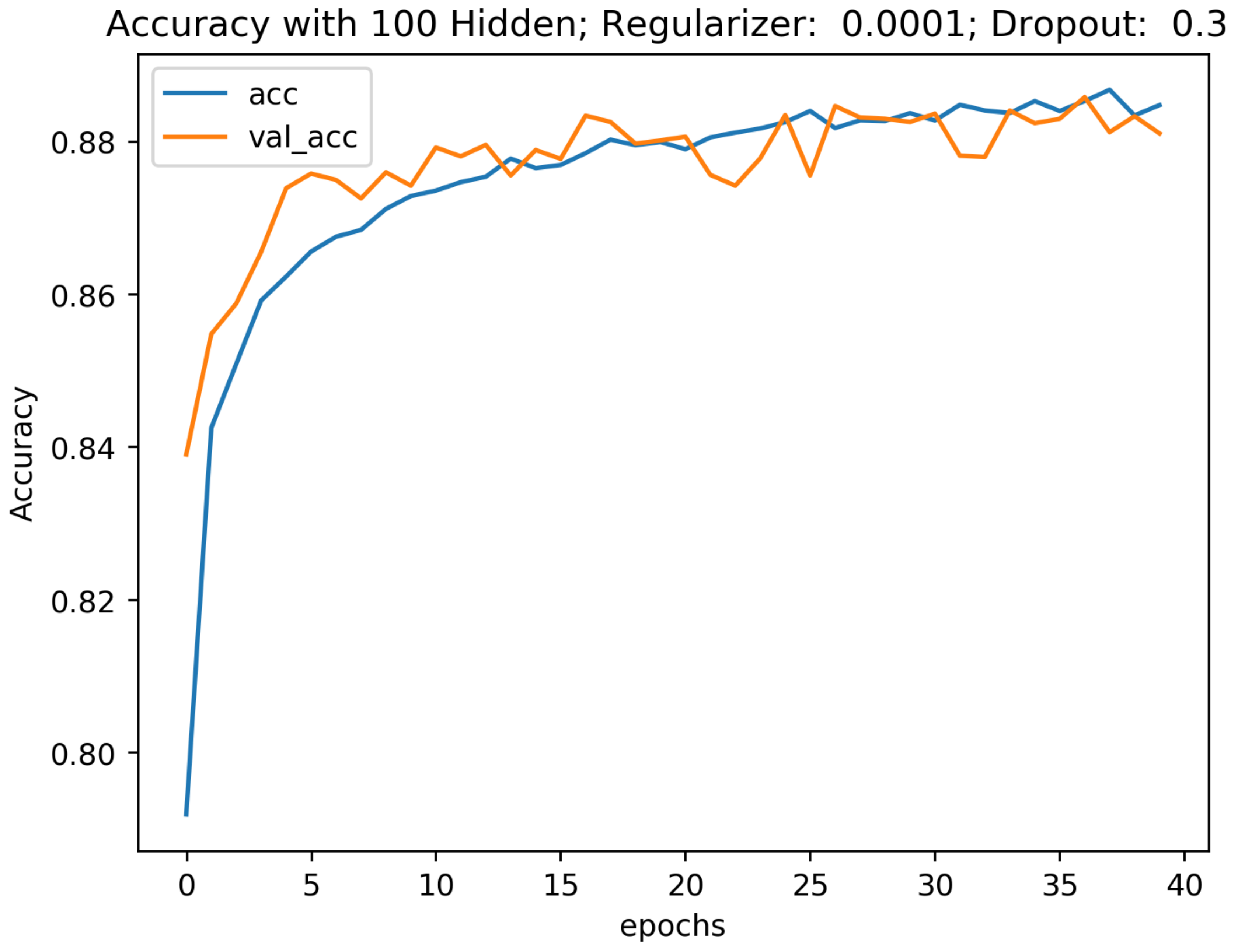
dropout rate ~ 20%

L2 Regularization: [1e-5, 1e-3]

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

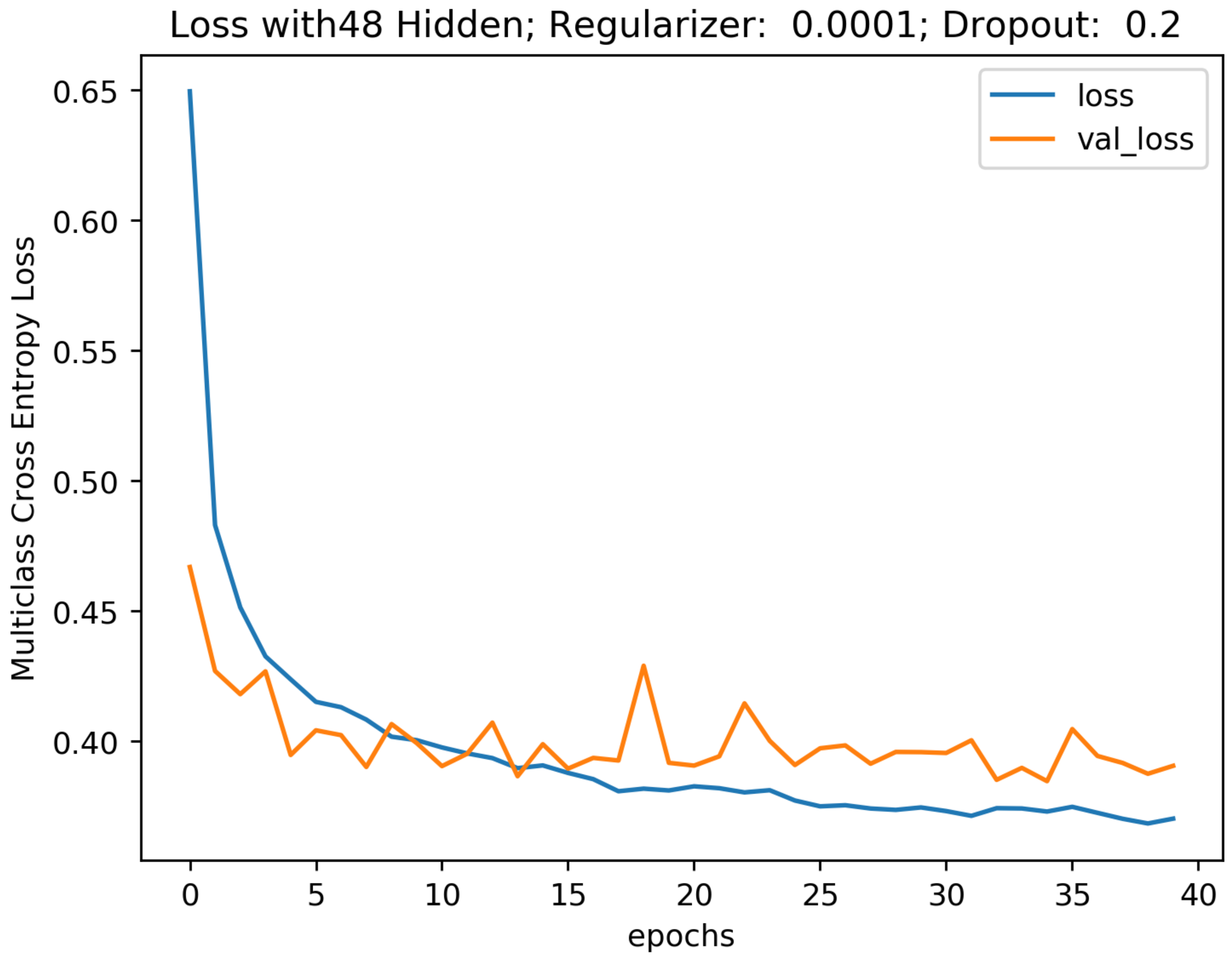
Smaller Model, Less Regularization

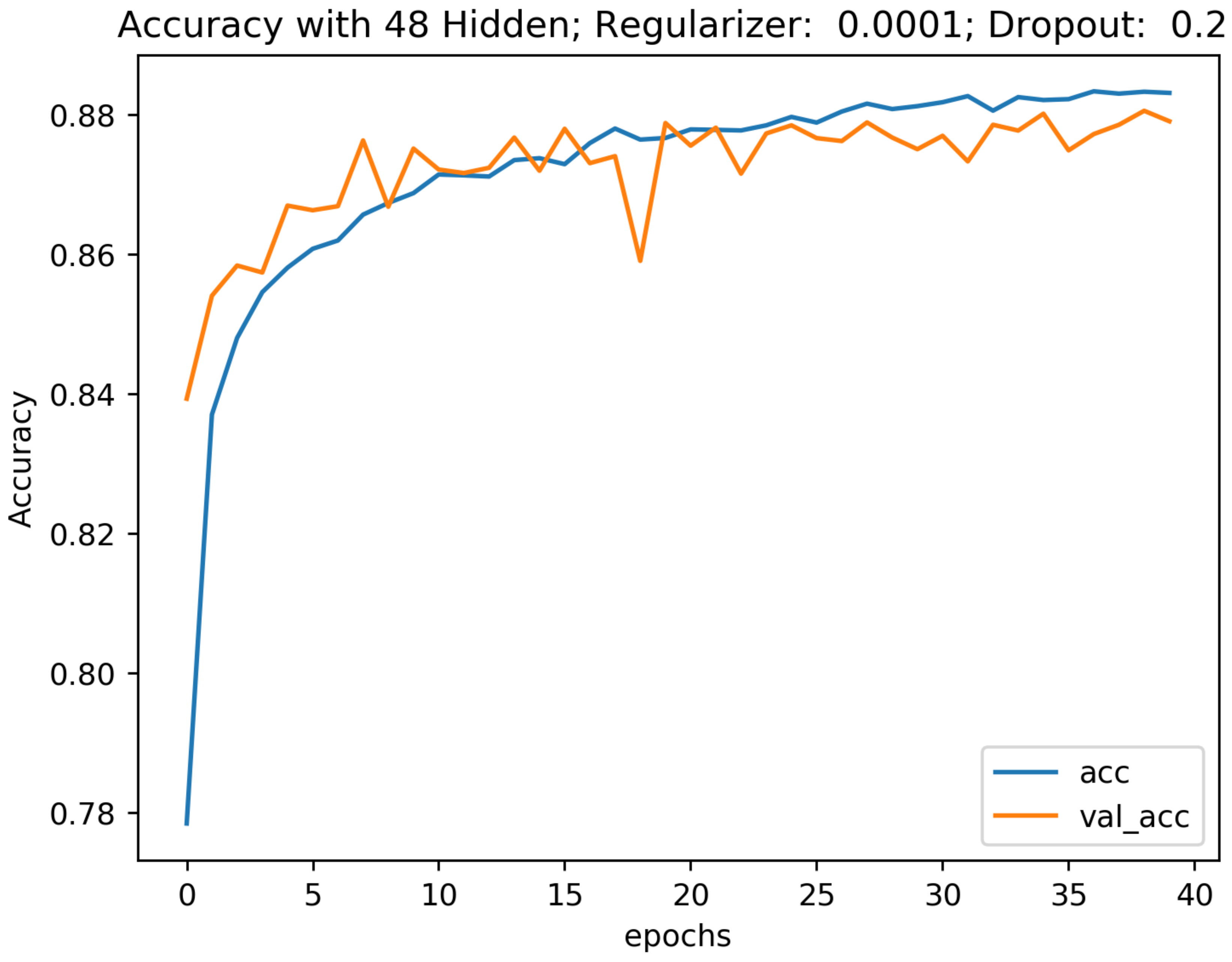




similar results with 100 hidden neurons

Smaller Model, Less Regularization



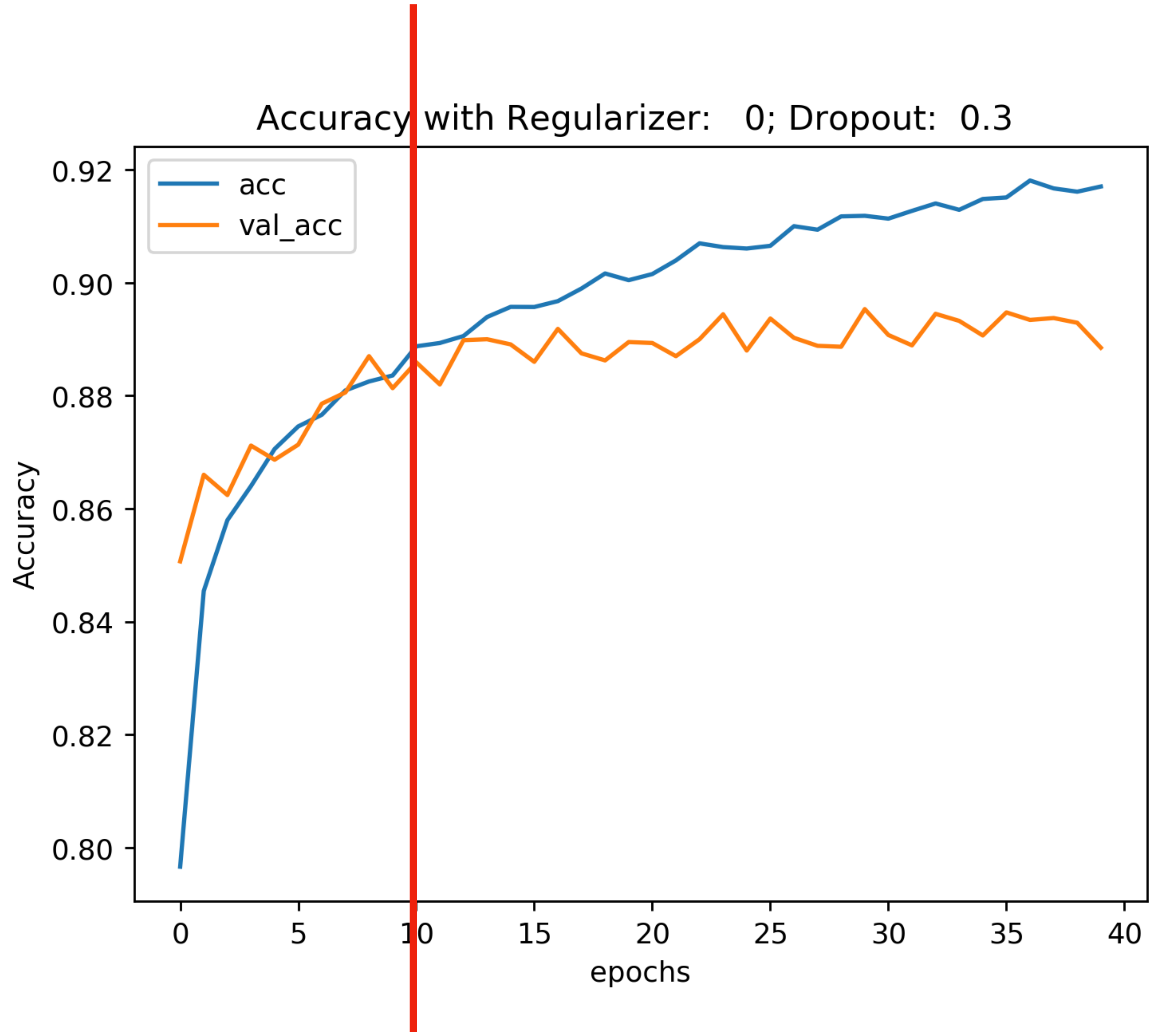


similar results with 48 hidden neurons

Another Regularization Method

“early stopping”

just stop when you val starts doing consistently better than your train



stop at ~10 epochs

Outline for Slides

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

Optimizers

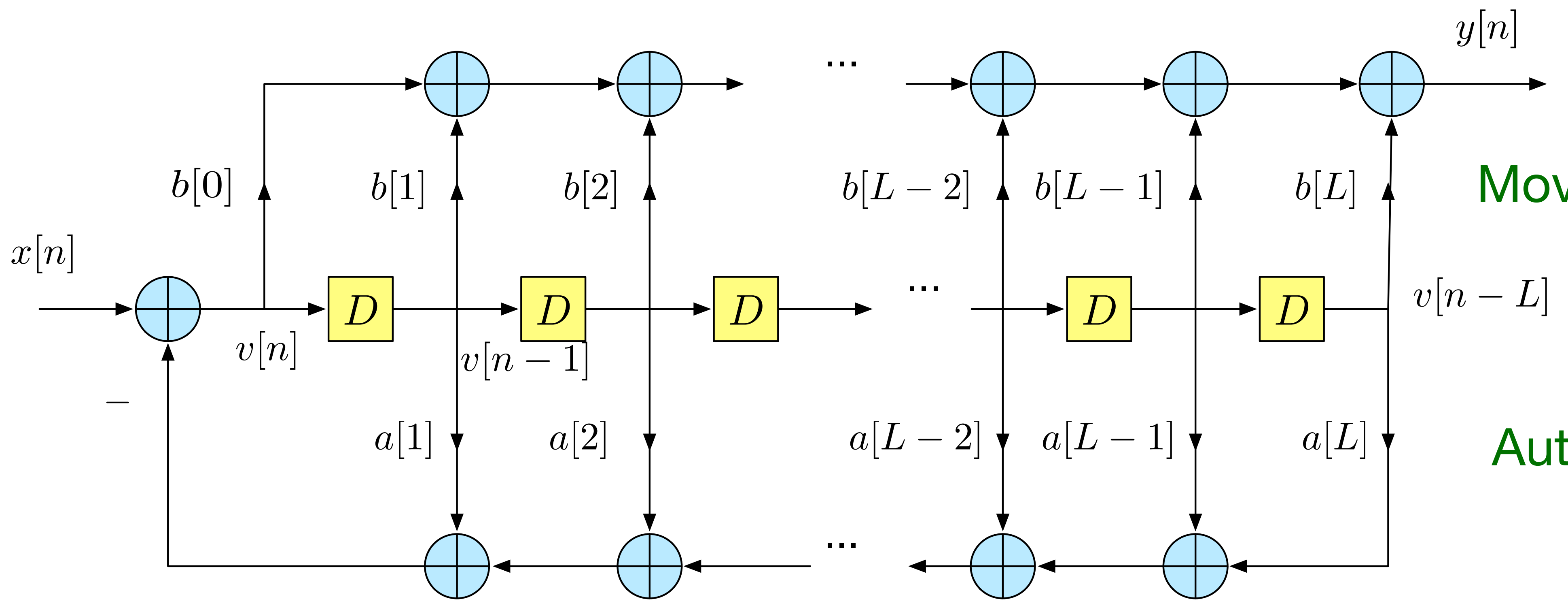
Optimizers are simply **modifications and tweaks** to the basic Stochastic Gradient Descent (SGD)

Main kinds of modifications:

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

1 and 2 are usually associated with the "optimizer" and the learning rate schedule is seen as a separate design task

Review of ARMA LTI Filters



Moving Average component

Autoregressive component

this is a canonical block diagram for an Lth order filter

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

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

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

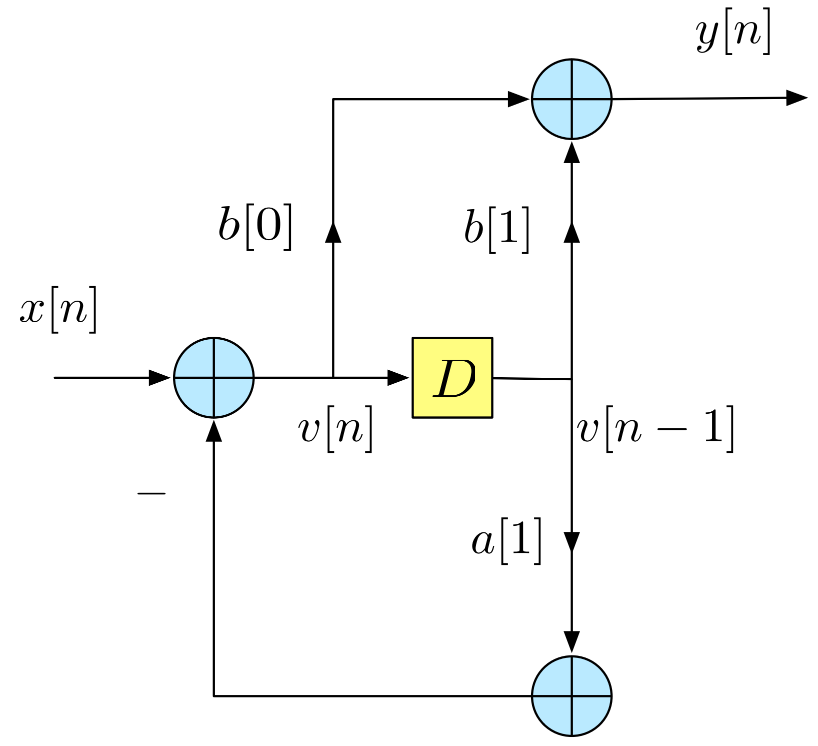
implements this difference equation:

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

Frequency response:

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

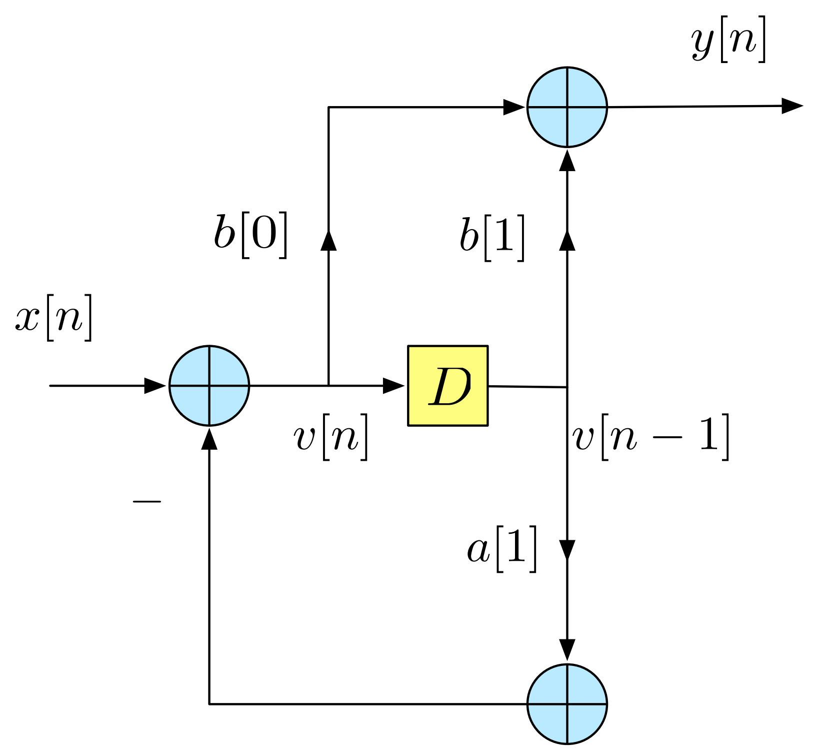
first order ARMA filter



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

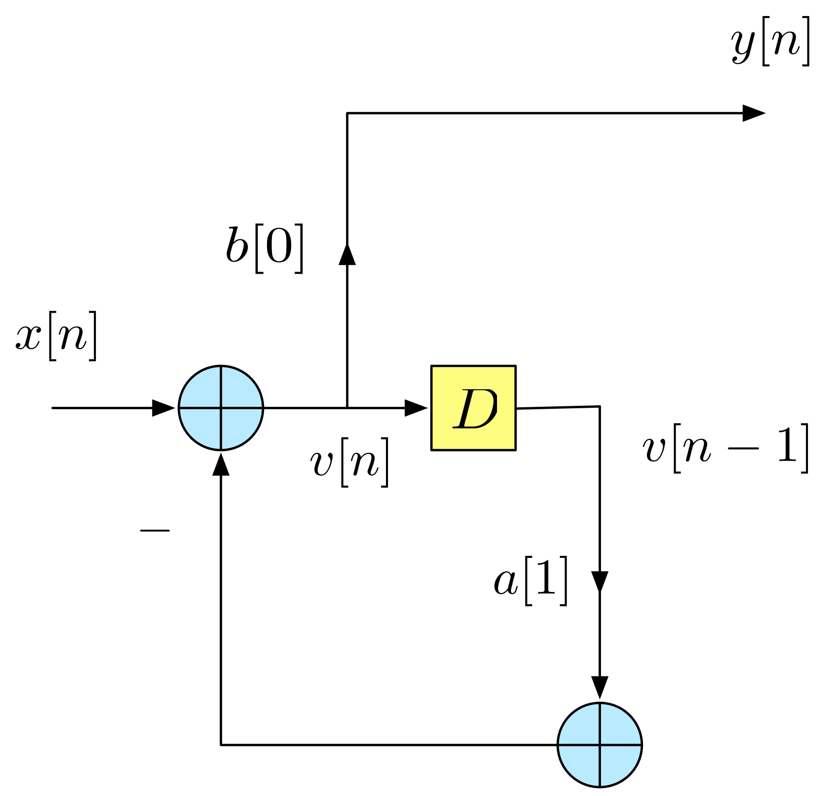
$$H(z) = \frac{b[0] + b[1]z^{-1}}{1 + a[1]z^{-1}}$$

Review of First Order LTI Filters



ARMA

One pole, one zero



AR

One pole

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

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

special cases for AR1:

Unit DC-Gain AR1:

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

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

this has input-gain = (1-alpha)

Unit input-Gain AR1:

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

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

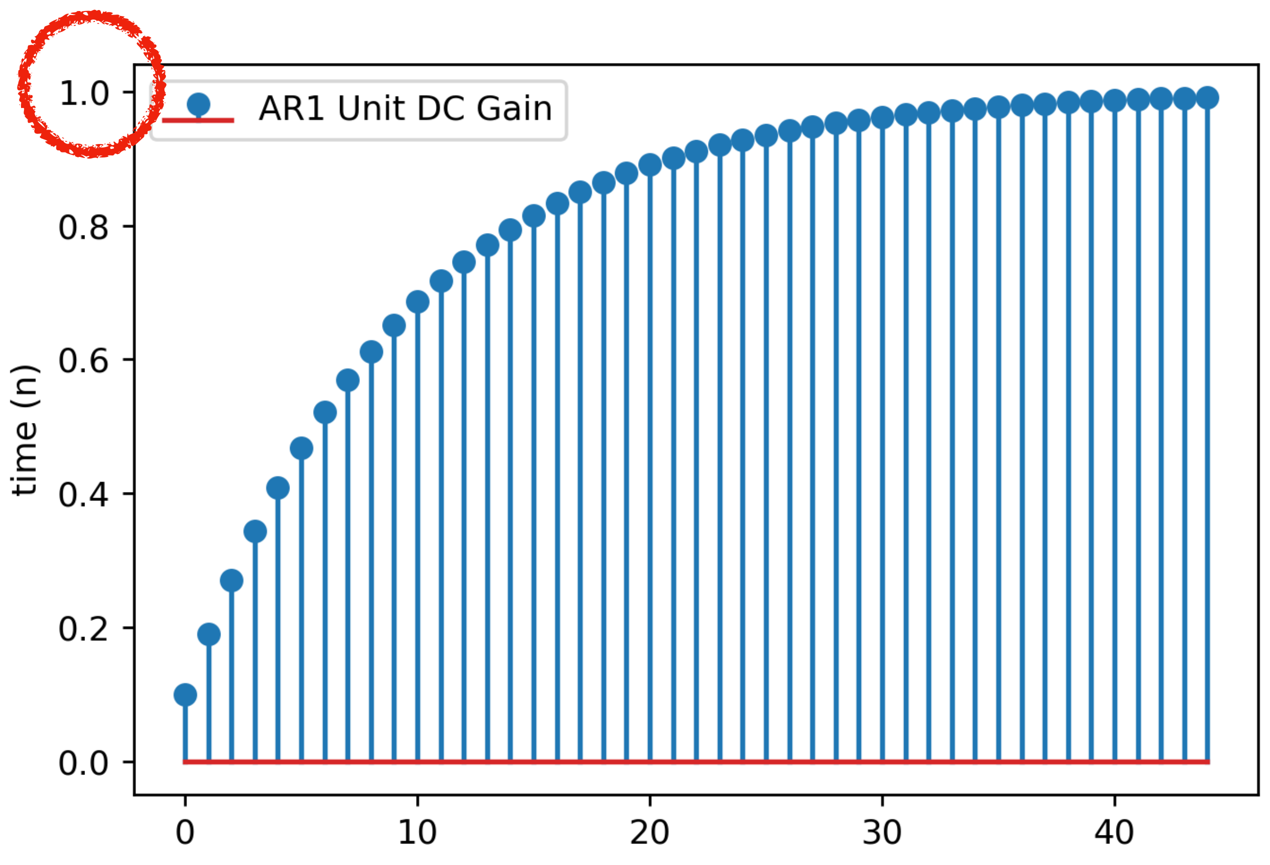
this has DC-gain = 1/(1-alpha)

Recall: as alpha approaches 1, the filter has more memory and becomes more low-pass

Review of First Order LTI Filters

special cases for AR1:

unit step response with alpha = 0.9



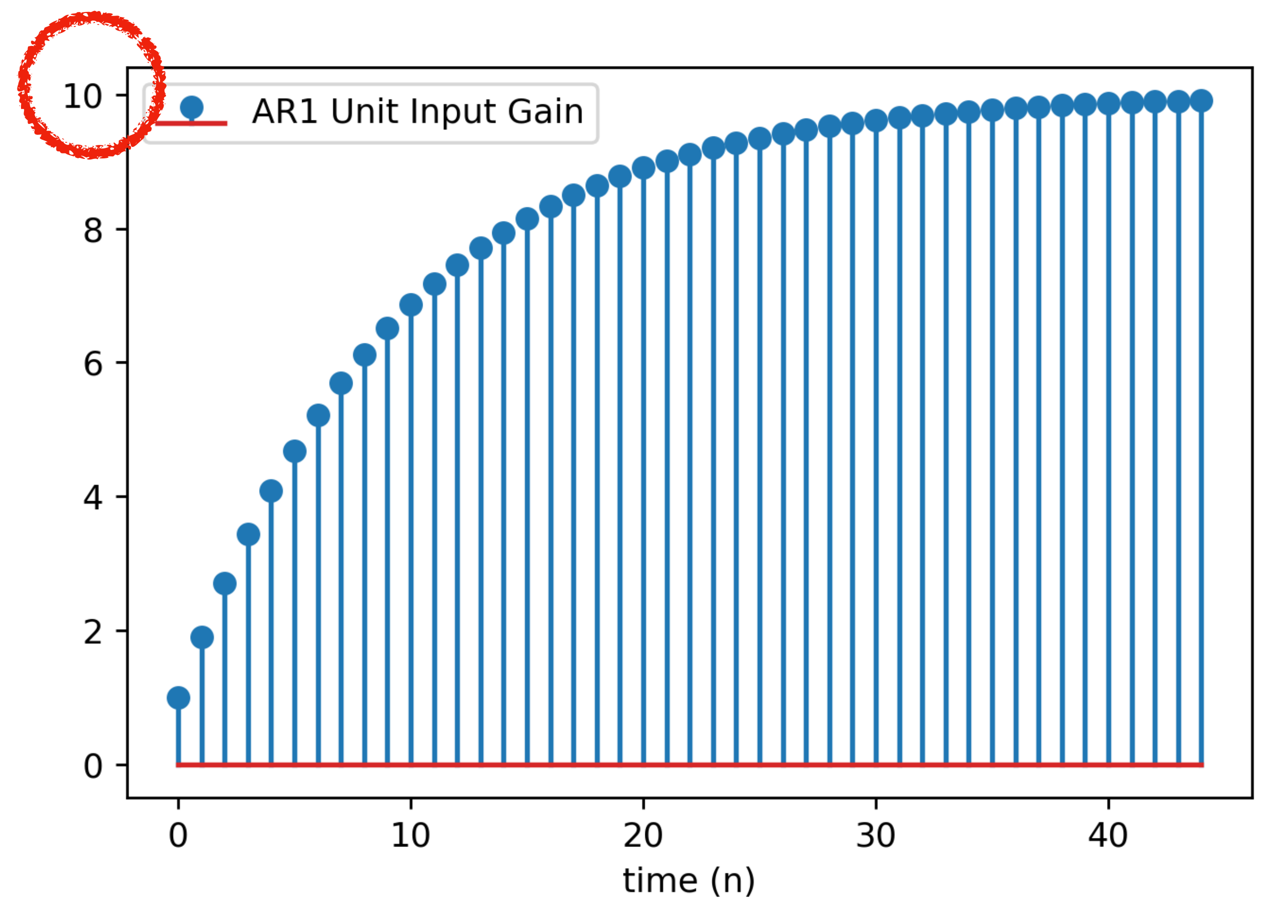
$$s[n] = 1 - \alpha^{n+1}$$

Unit DC-Gain AR1:

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

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

this has input-gain = (1-alpha)



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

Unit input-Gain AR1:

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

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

this has DC-gain = 1/(1-alpha)

Recall: as alpha approaches 1, the filter has more memory and becomes more low-pass

Transient Compensation

Unit input Gain AR1: pole dependent DC gain

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

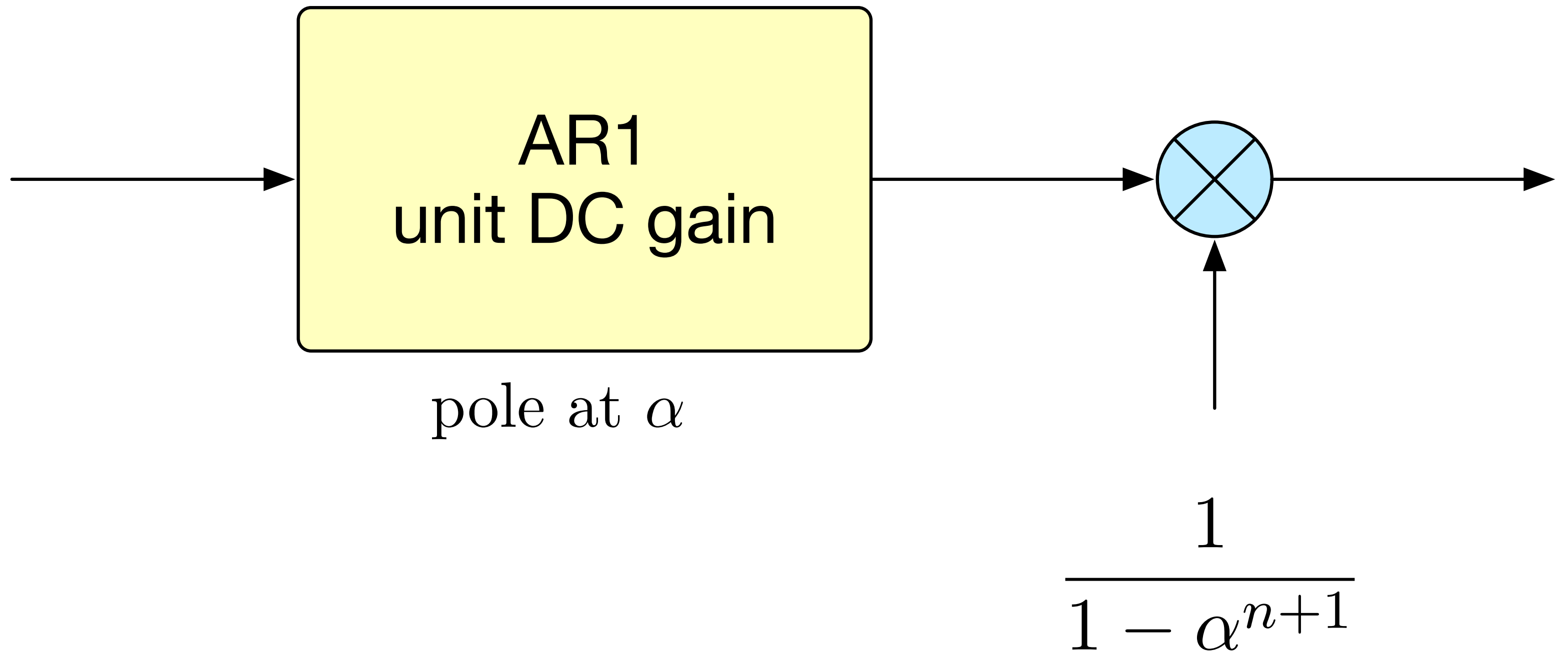
Unit DC-Gain AR1:

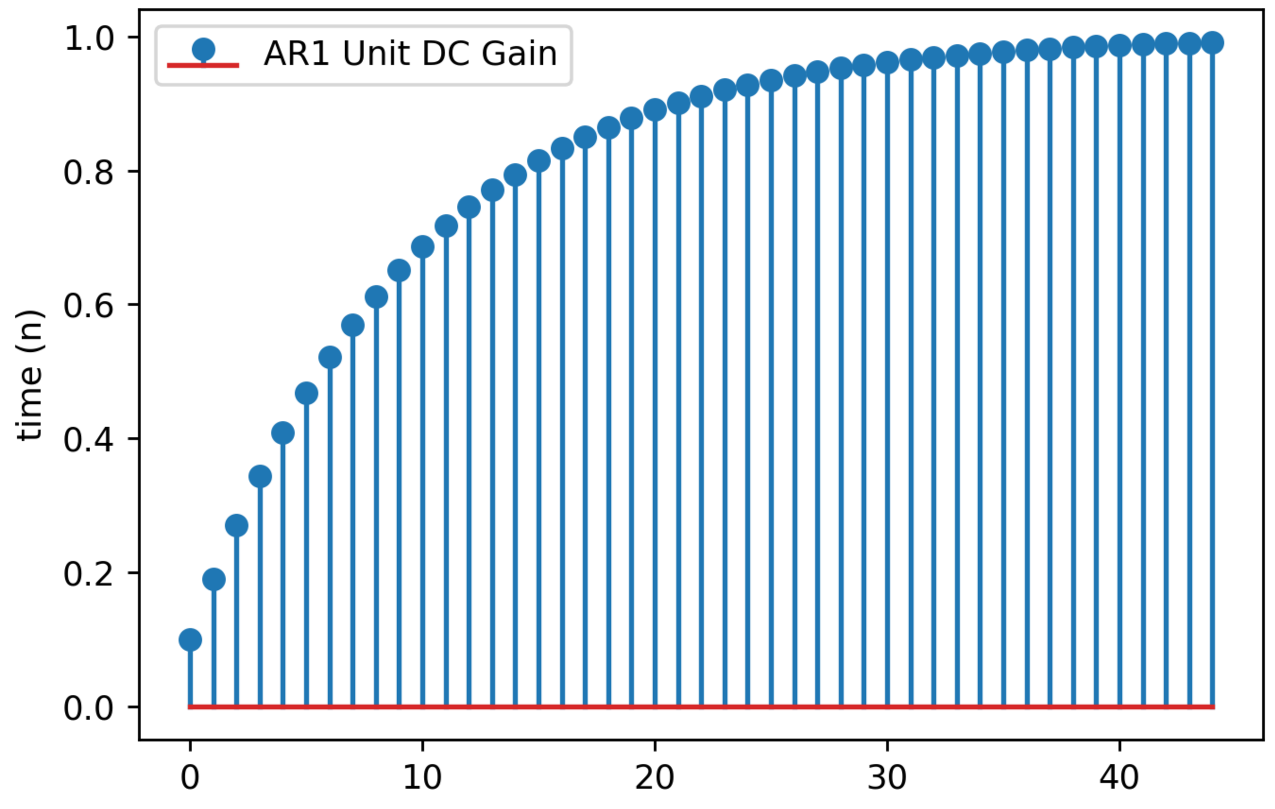
$$y[n] = \alpha y[n-1] + (1-\alpha)x[n]$$

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

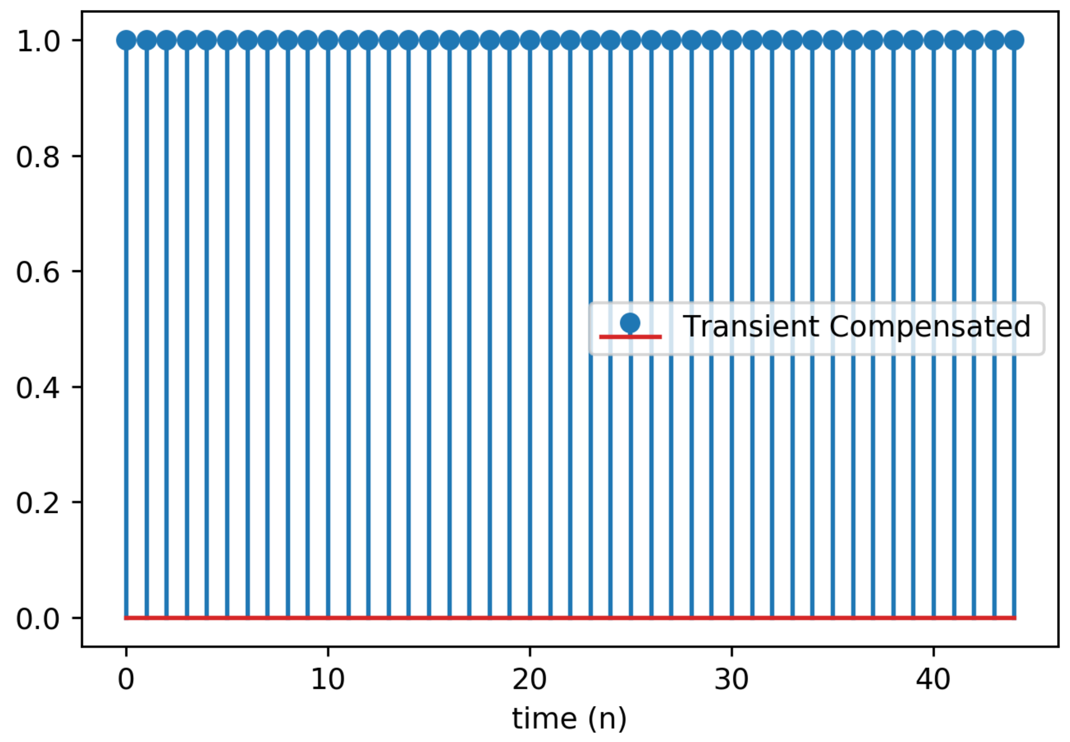
$$s[n] = 1 - \alpha^{n+1}$$

transient compensation





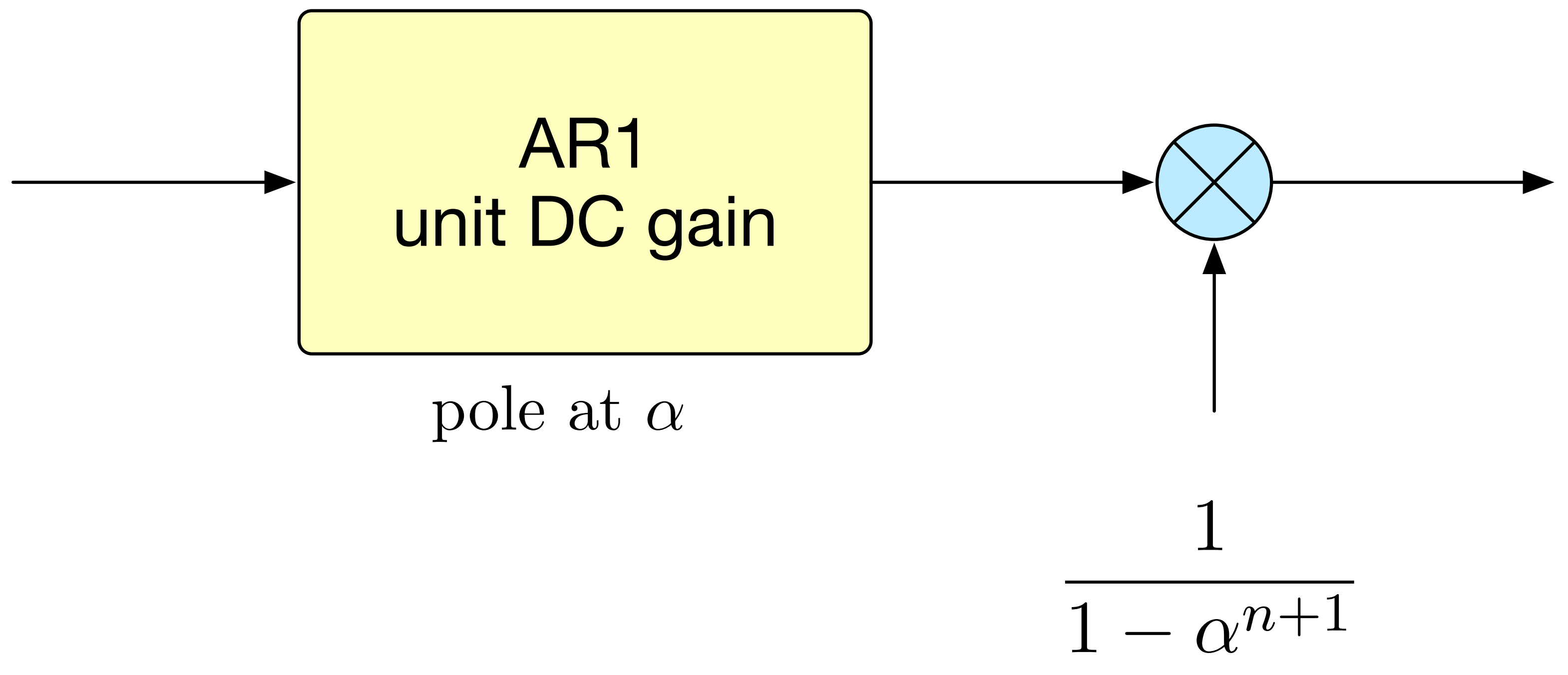
transient compensated step response

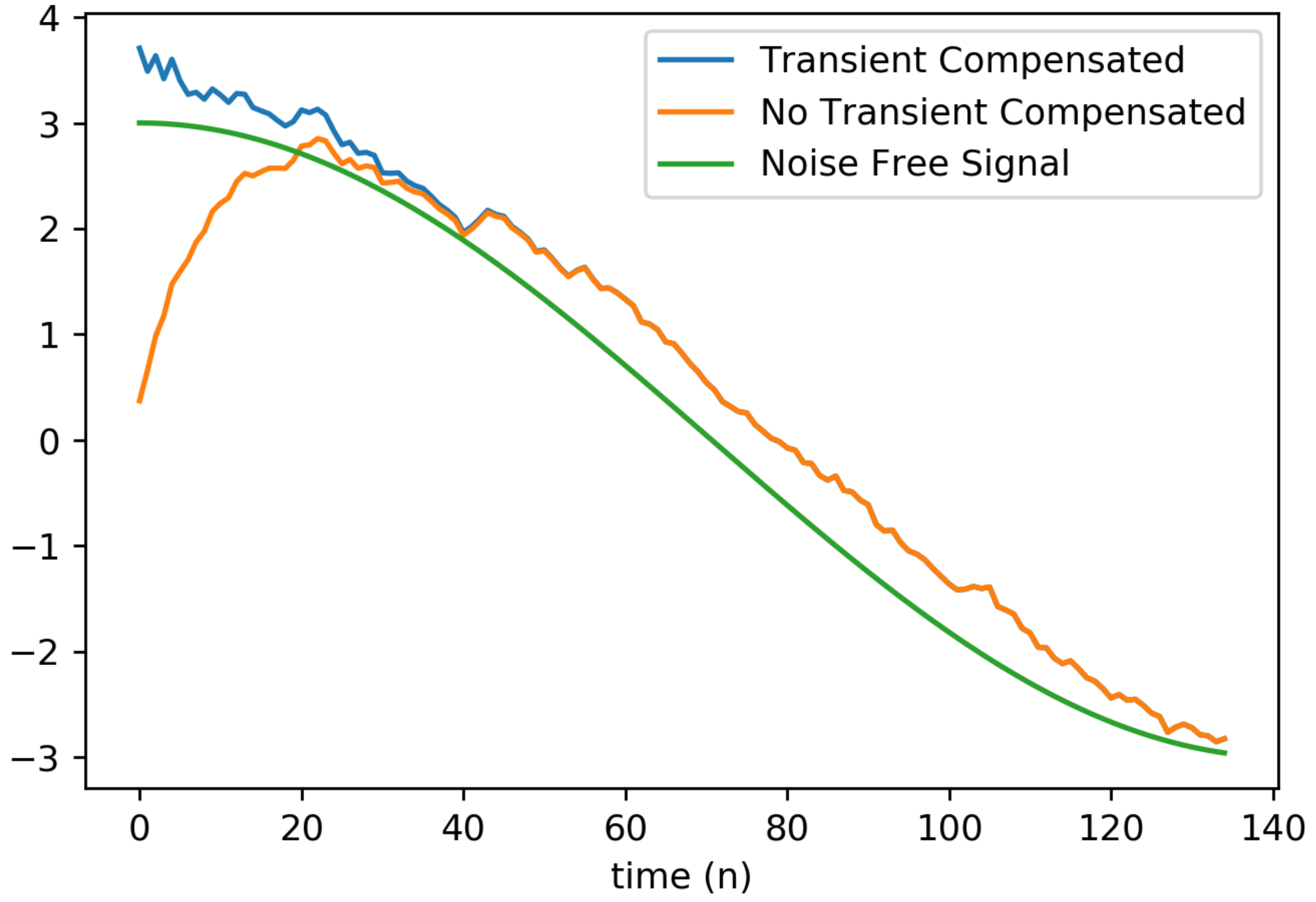


this works for any scaled step input!

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)

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).

General Optimizer Structure + SDG

i ~ indexes parameter updates (mini-batches)

parameter update:

$$\boxed{\theta[i] = \theta[i-1] + \Delta[i]}$$

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:

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

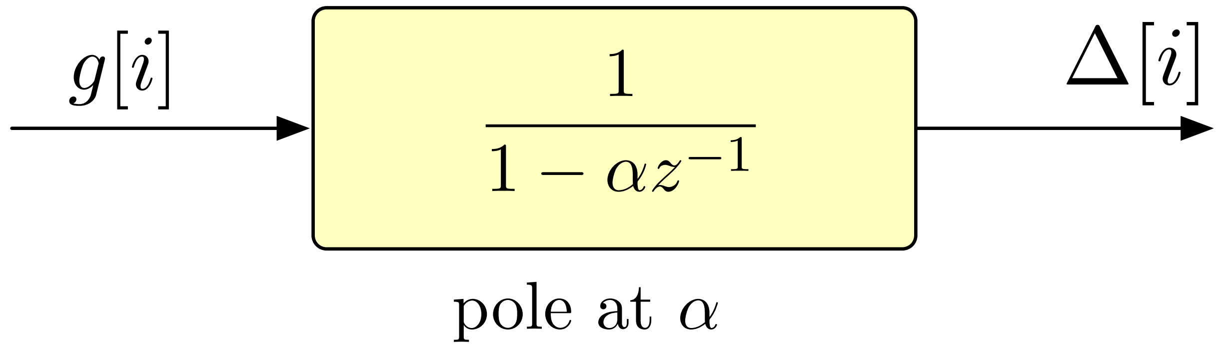
SGD with momentum:

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

v is called the “velocity”

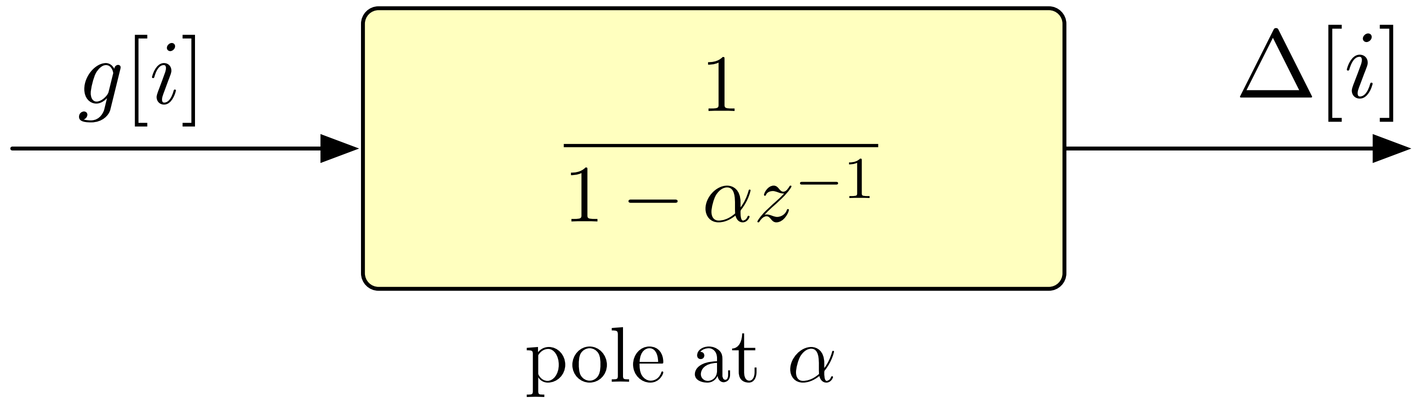
alpha is called the “momentum”

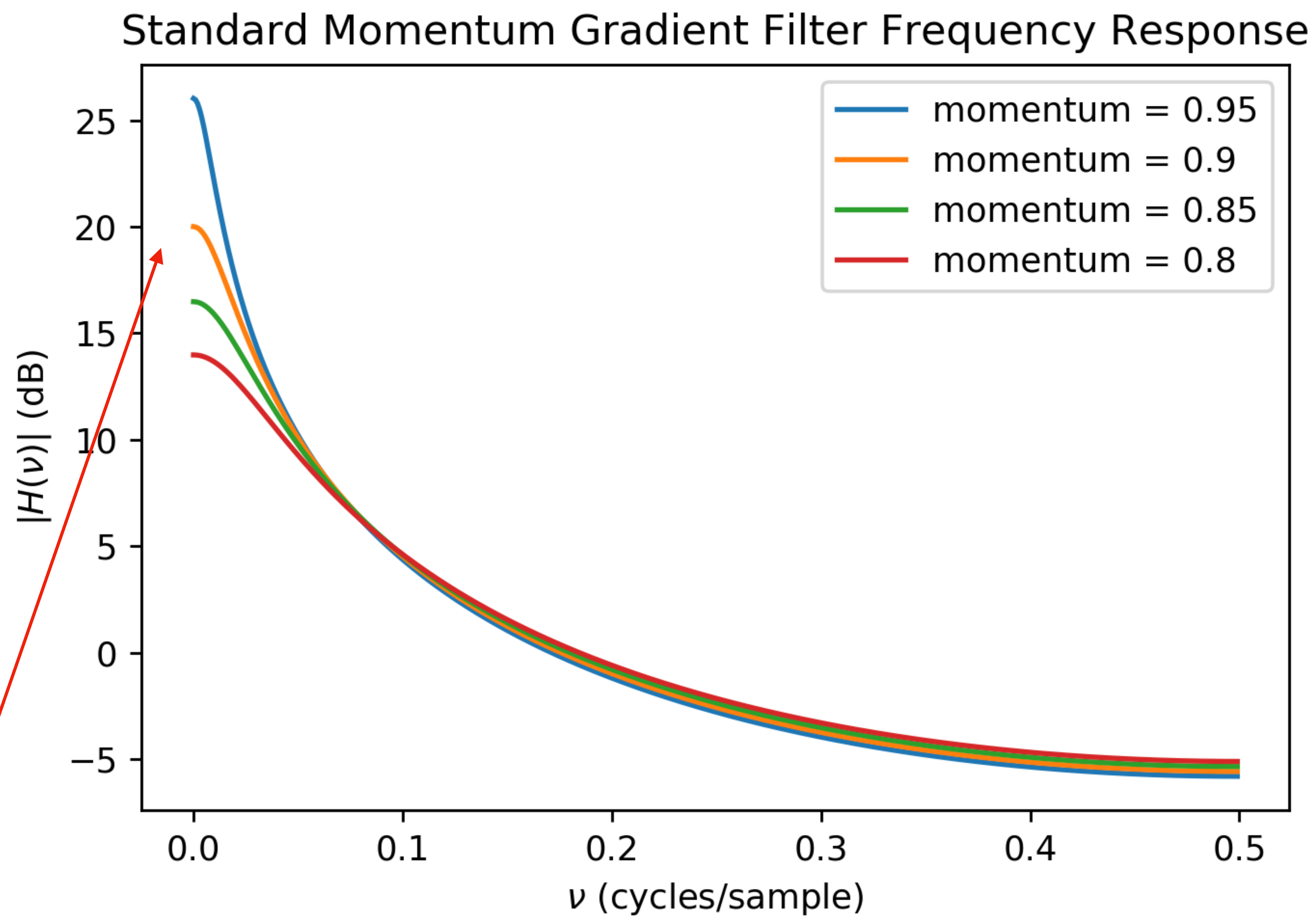
(alpha ~ 0.9)



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

(standard) Momentum





Momentum: low-pass filter on the gradient — removes high-free 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: $\boxed{\theta[i] = \theta[i-1] + \Delta[i]}$

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

SGD with Nesterov momentum:

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

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

pole at α

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

v is called the “velocity”

alpha is called the “momentum”

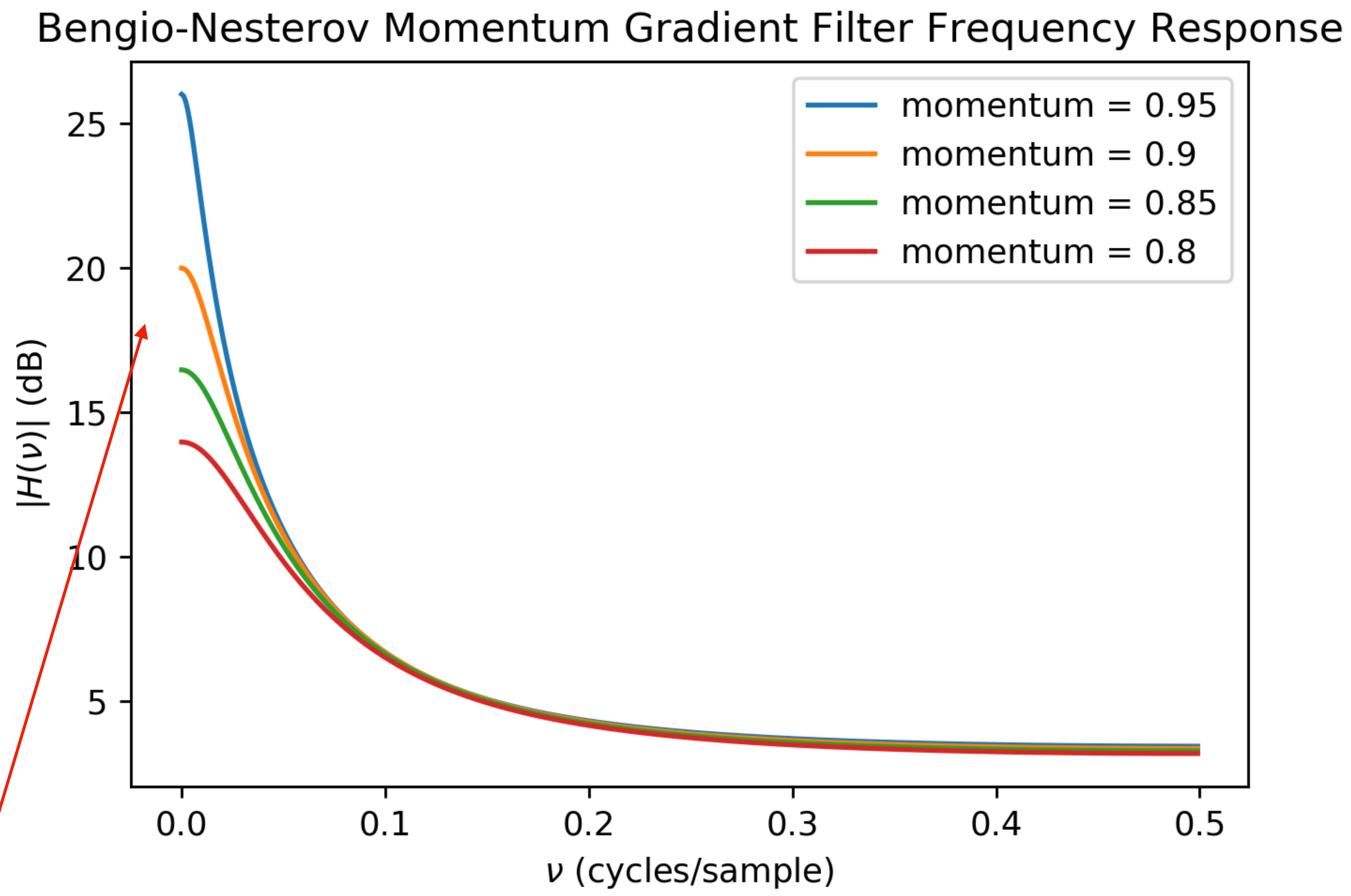
(alpha ~ 0.9)

SGD with Nesterov Momentum

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

pole at α

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



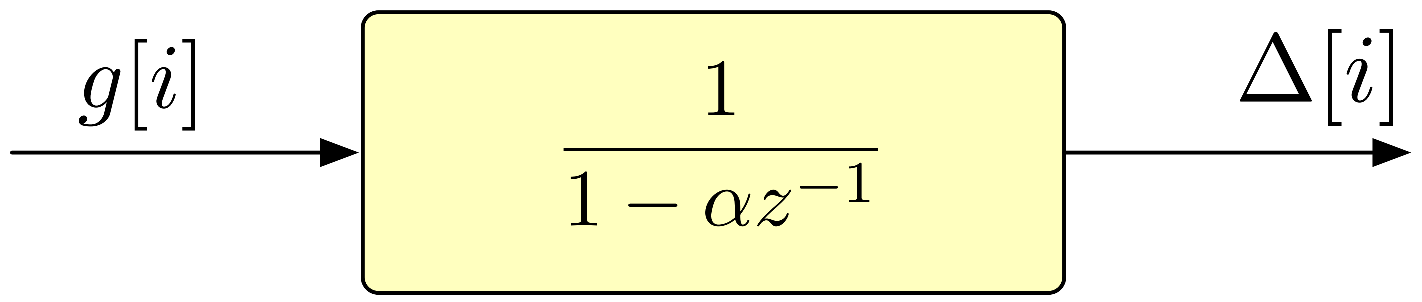
Momentum: low-pass filter on the gradient — removes high-free gradient noise

note that your momentum and learning rate are coupled

choosing larger momentum, effectively increases your learning rate

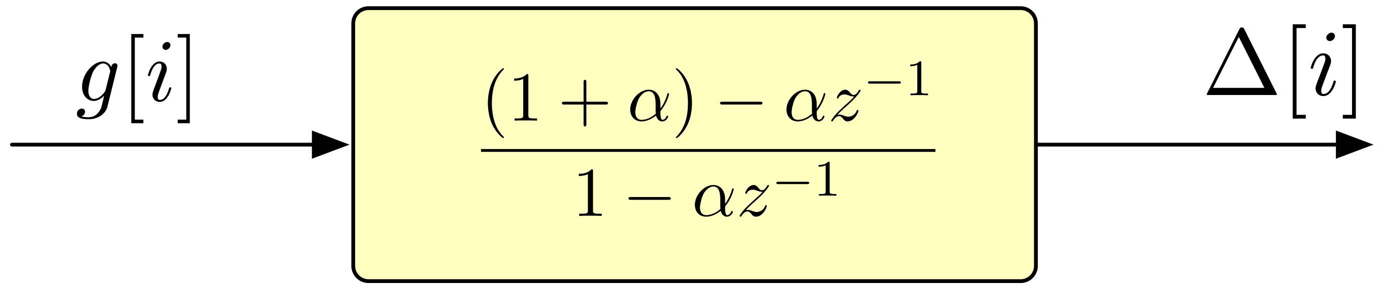
Standard Momentum vs Nesterov Momentum

standard



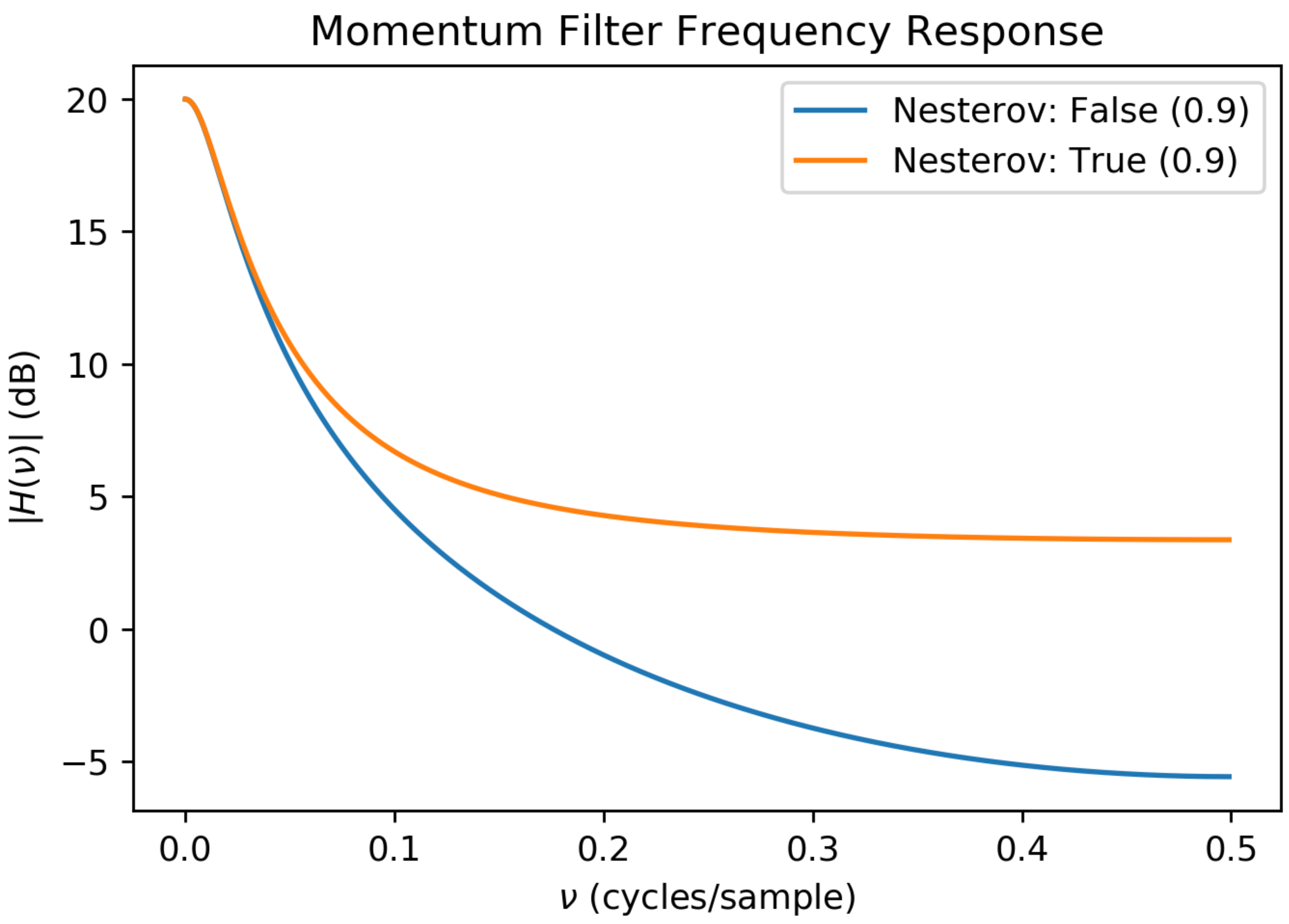
pole at α

Nesterov



pole at α

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



Nesterov does not attenuate the high frequencies as much as standard momentum

Nesterov Momentum (typical motivation)

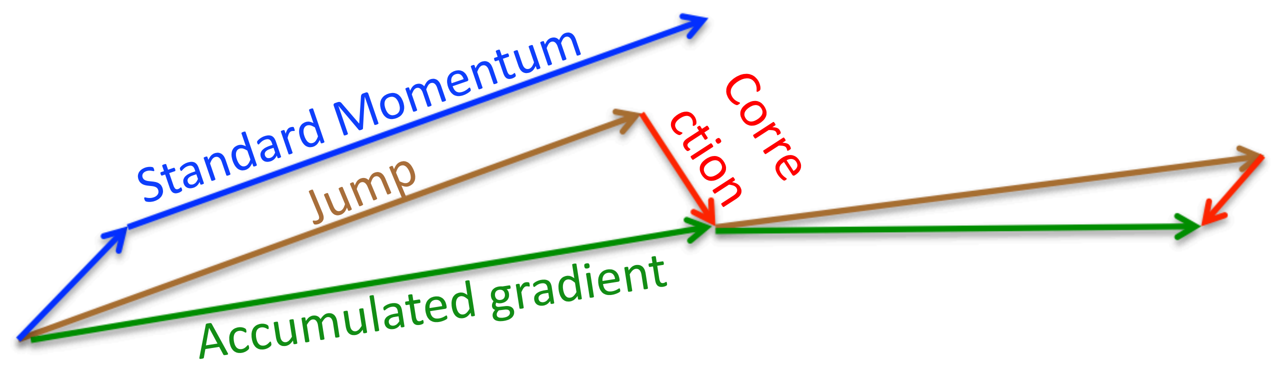
Usually motivated as doing a "preliminary" parameter update based before updating velocity and then adjusting for velocity update

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

$$\theta_t = \theta_{t-1} + v_t \quad (2)$$

what exactly is this?!?

typical explanation



Geoffrey Hinton's slides: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

$$v_t = \mu_{t-1} v_{t-1} - \epsilon_{t-1} \nabla f(\Theta_{t-1}) \quad (6)$$

$$\begin{aligned} \Theta_t &= \Theta_{t-1} - \mu_{t-1} v_{t-1} + \mu_t v_t + v_t \\ &= \Theta_{t-1} + \mu_t \mu_{t-1} v_{t-1} - (1 + \mu_t) \epsilon_{t-1} \nabla f(\Theta_{t-1}) \quad (7) \end{aligned}$$

"Bengio's Formulation"

(this is what tf.keras does)

best references of this type I could find (still confusing!):

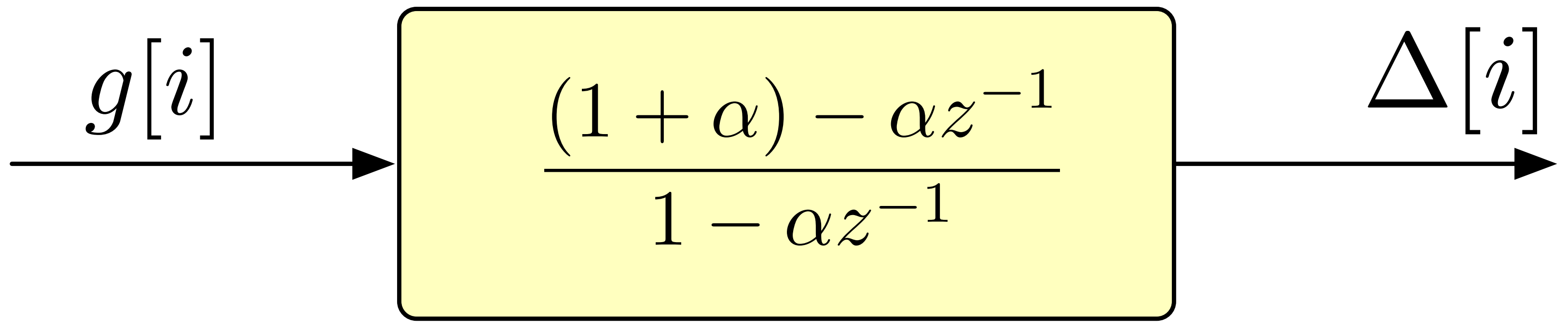
Bengio, Yoshua, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in optimizing recurrent networks." *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013.

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

Nesterov Momentum

“Bengio’s Formulation”

$$
\begin{aligned}
v[i] &= \alpha v[i-1] + g[i] \\
\theta[i] &= \theta[i-1] + (1+\alpha)v[i] - \alpha v[i-1] \\
\Delta[i] &= (1+\alpha)v[i] - \alpha v[i-1] \\
&= v[i] + \alpha \underbrace{\left(v[i] - v[i-1]\right)}_{\sim \text{ acceleration}}
\end{aligned}
$$



pole at α

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

this formulation makes the pattern clear and one could choose any low-pass filter for this task — i.e., optimize a second order ARMA filter (e.g., Butterowrth)

Gradient Normalization

Basic Idea: estimate the RMS value of the gradient and normalize by that value

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

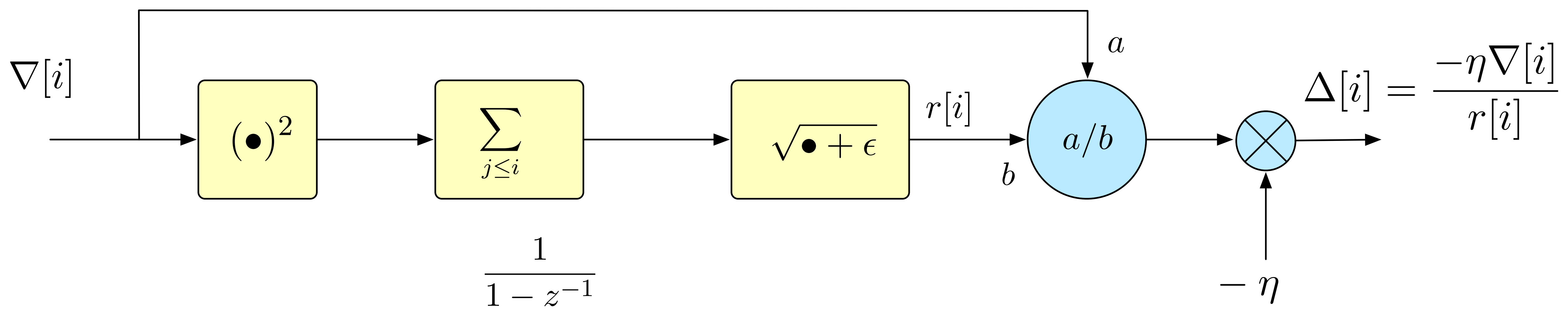
input step/gradient (update): $\nabla[i] = \frac{\partial C}{\partial \theta[i-1]}$ $\quad g[i] = -\eta \frac{\partial C}{\partial \theta[i-1]}$

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

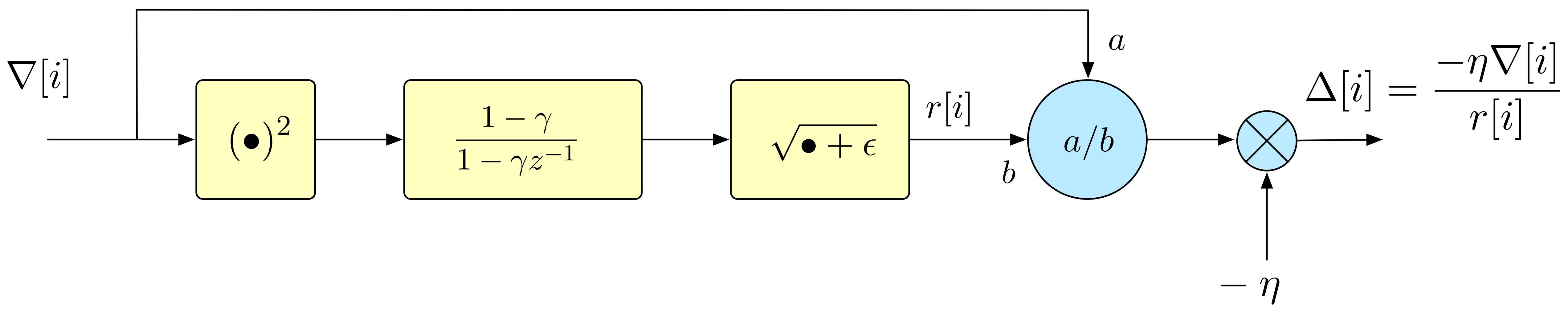
this is done by using some kind of low-pass filter on the the square of these quantities — i.e., like computing the sample second moment

Gradient Normalization Examples

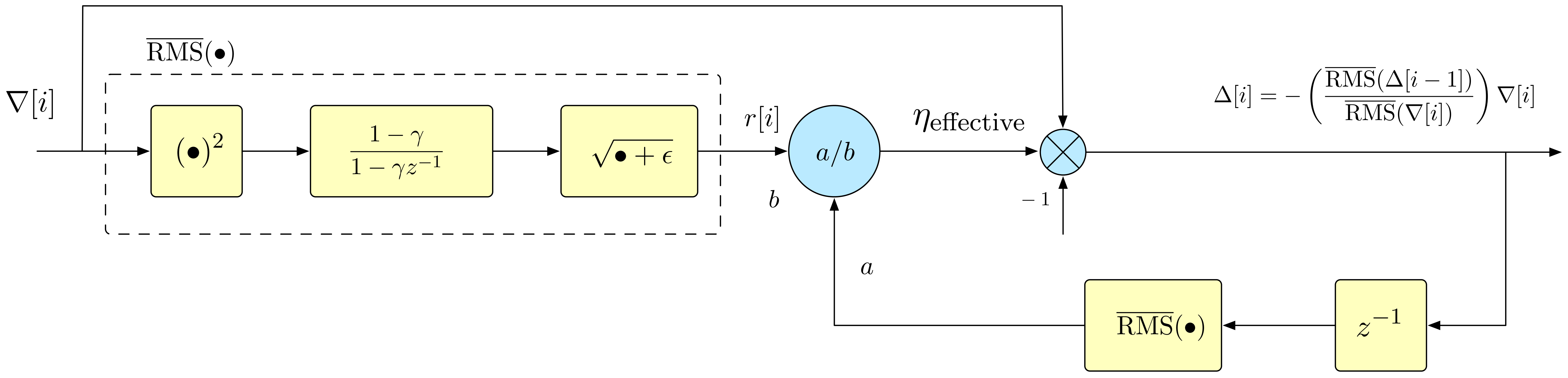
Adagrad:



RMSprop:



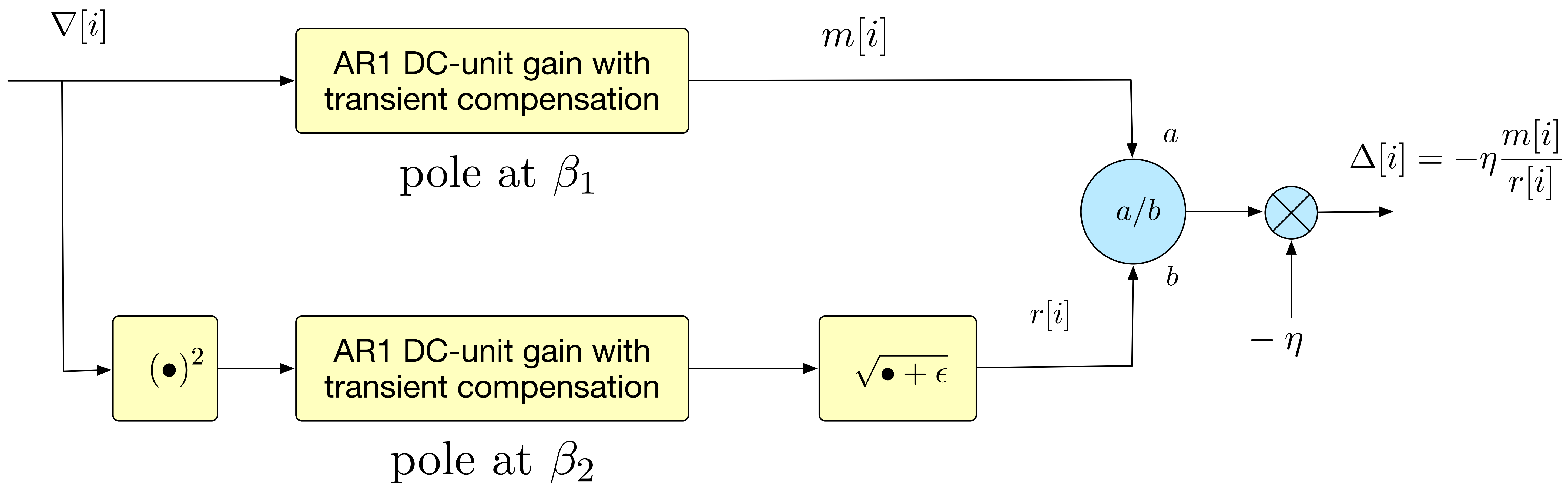
Adadelta:



Adam (the best of all worlds?)

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

use transient compensation to reduce the start-up effects on these filters



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.

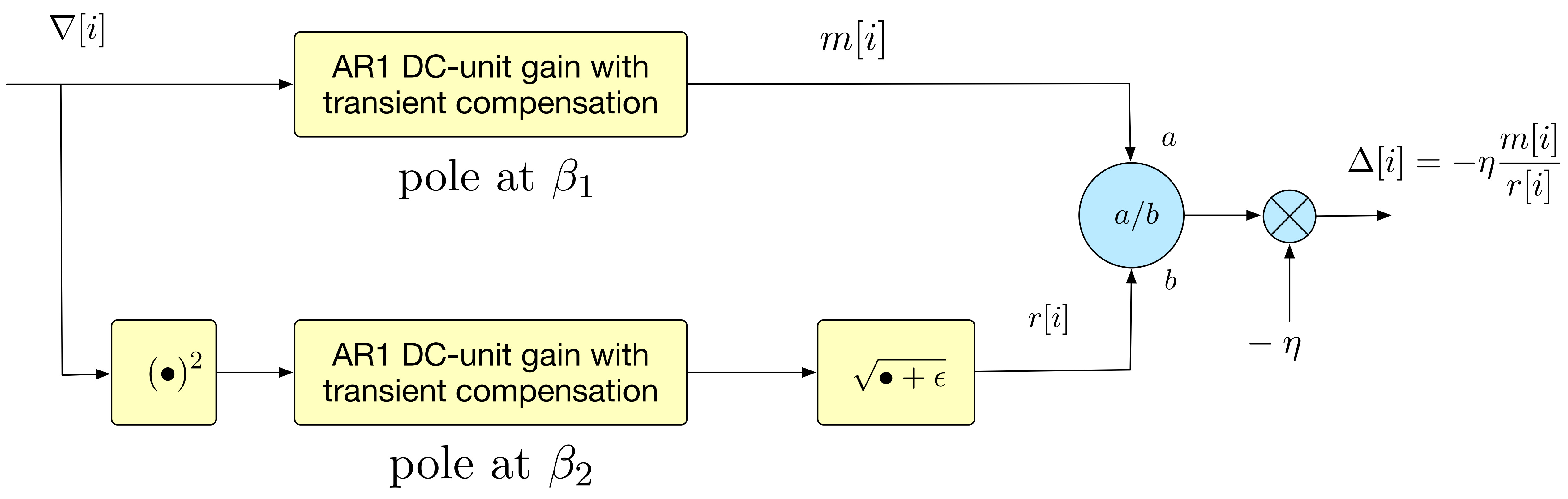
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd 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

Note: t starts from 1, I use i starting from 0

Adam in tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam




```
__init__(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name='Adam',
    **kwargs
)
```

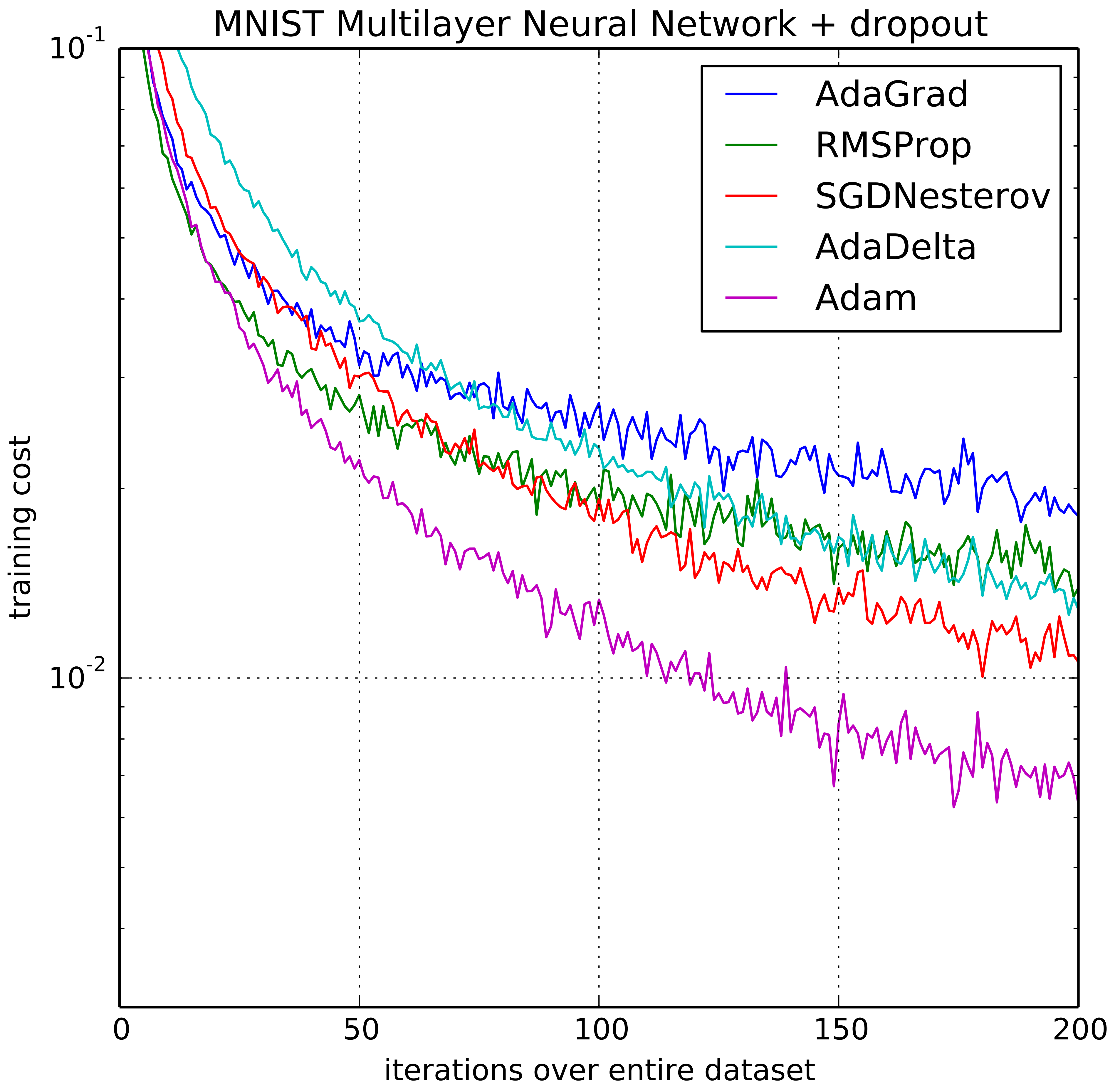
Contents

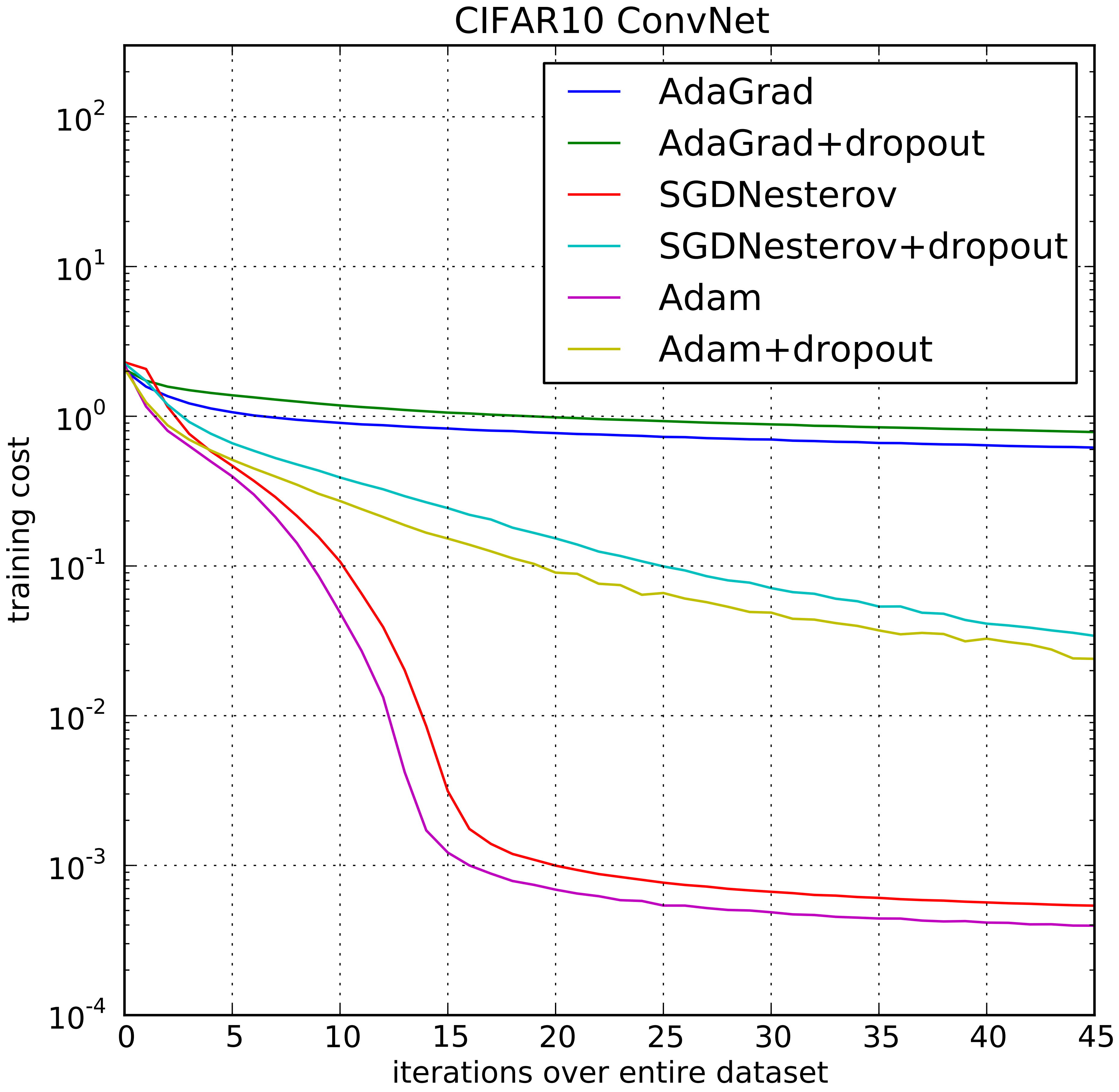
example:

```
my_adam = tf.keras.optimizers.adam(learning_rate=0.002, beta_1=0.92, beta_2=0.99, epsilon=1e-09)
our_first_model.compile(optimizer=my_adam, loss=SparseCategoricalCrossentropy(), metrics=['accuracy'])
```


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

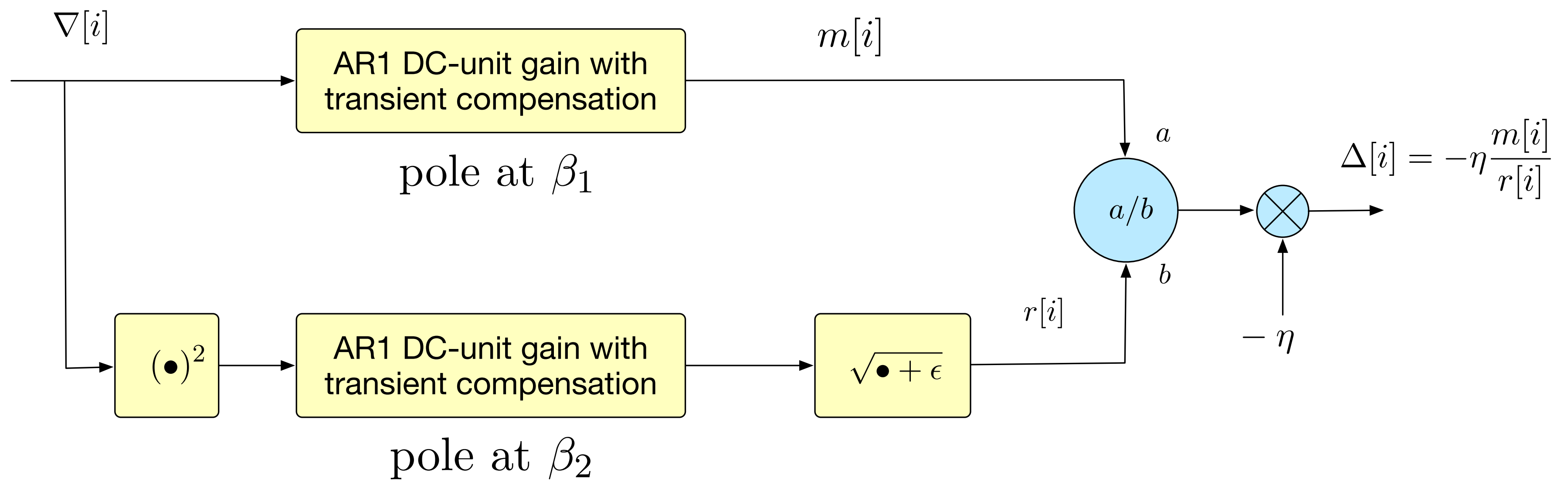
Adam Performance



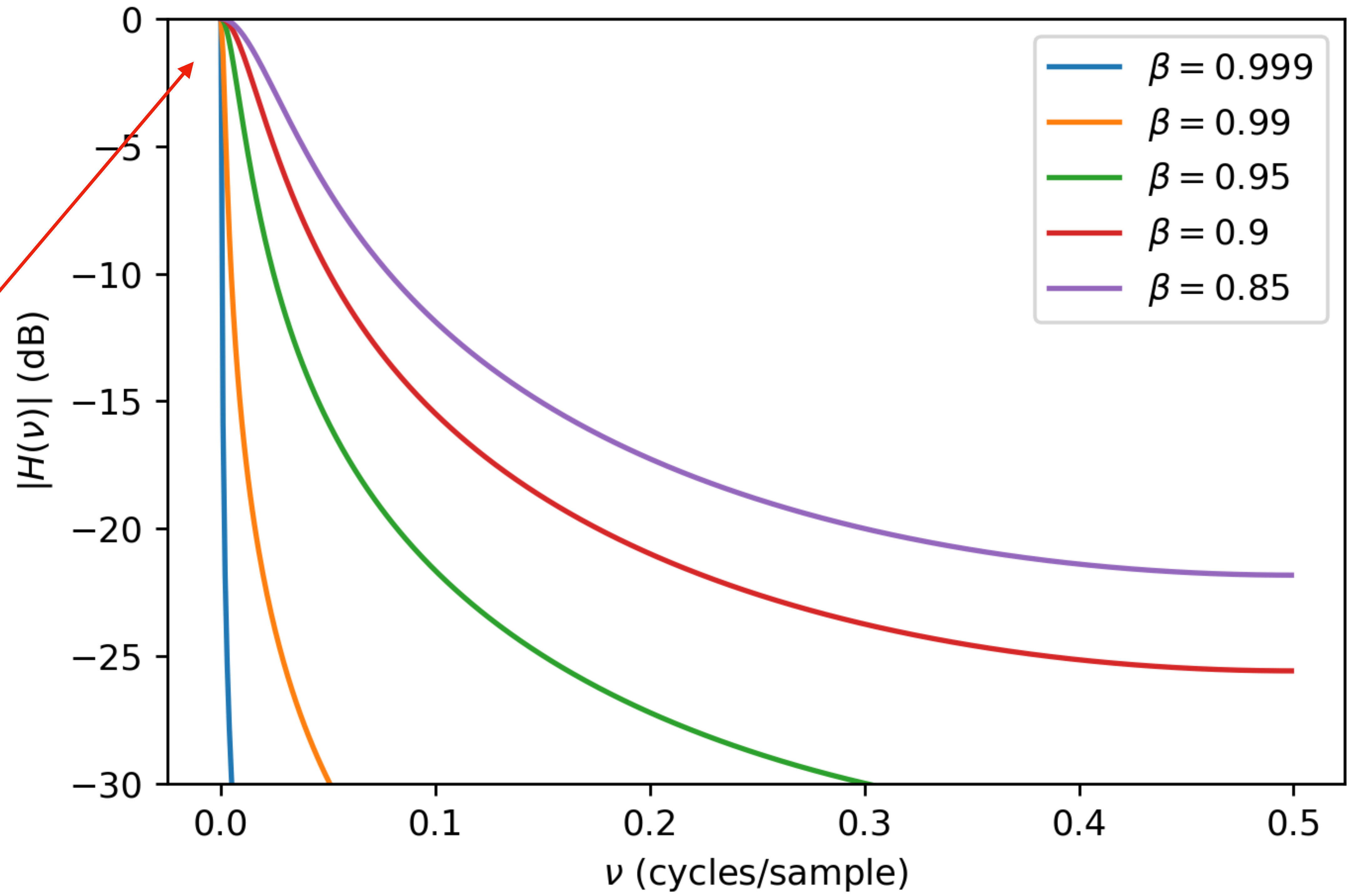


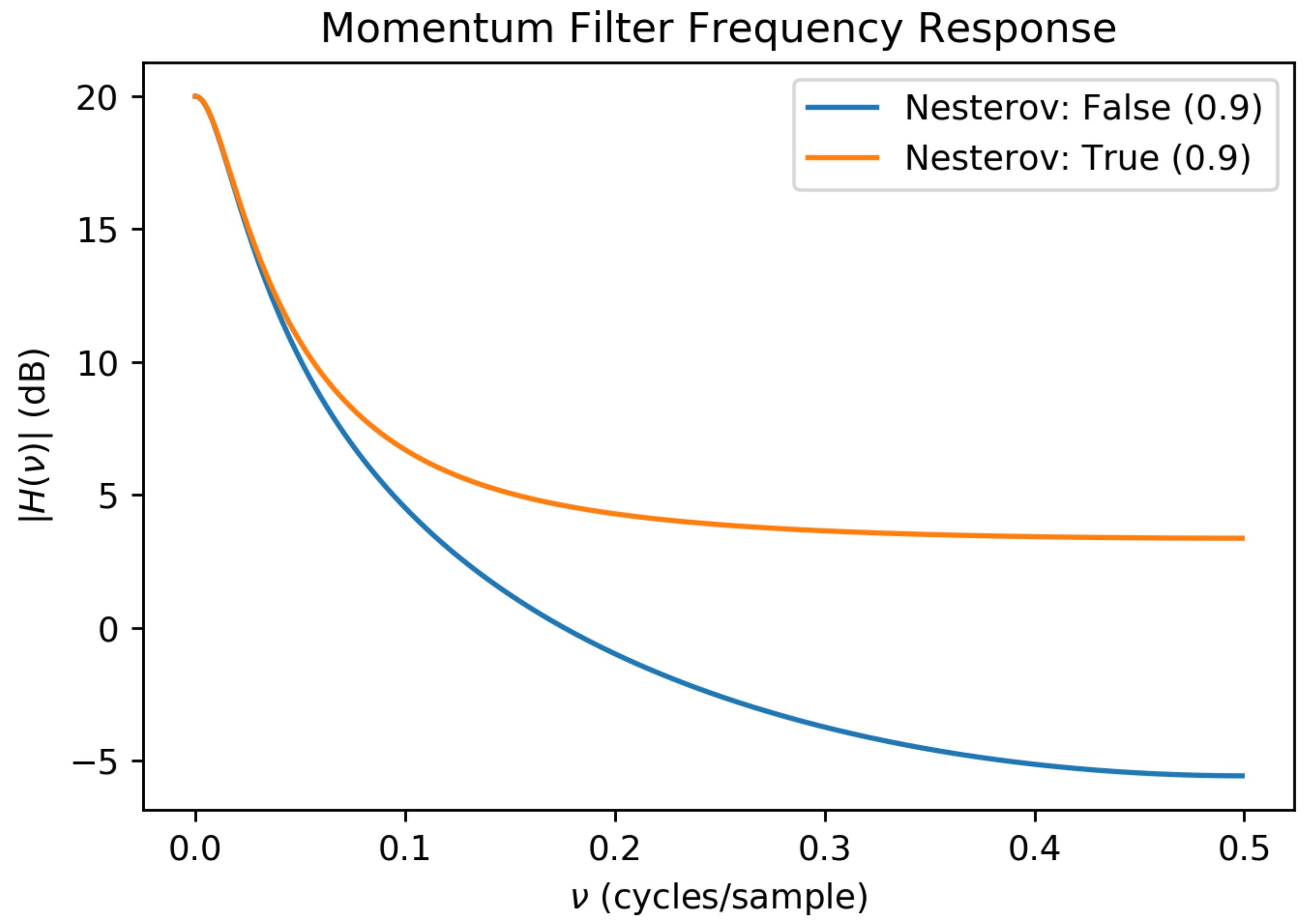
D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

Adam Gradient Filter Frequency Response



note that your momentum and learning rate are **not** coupled



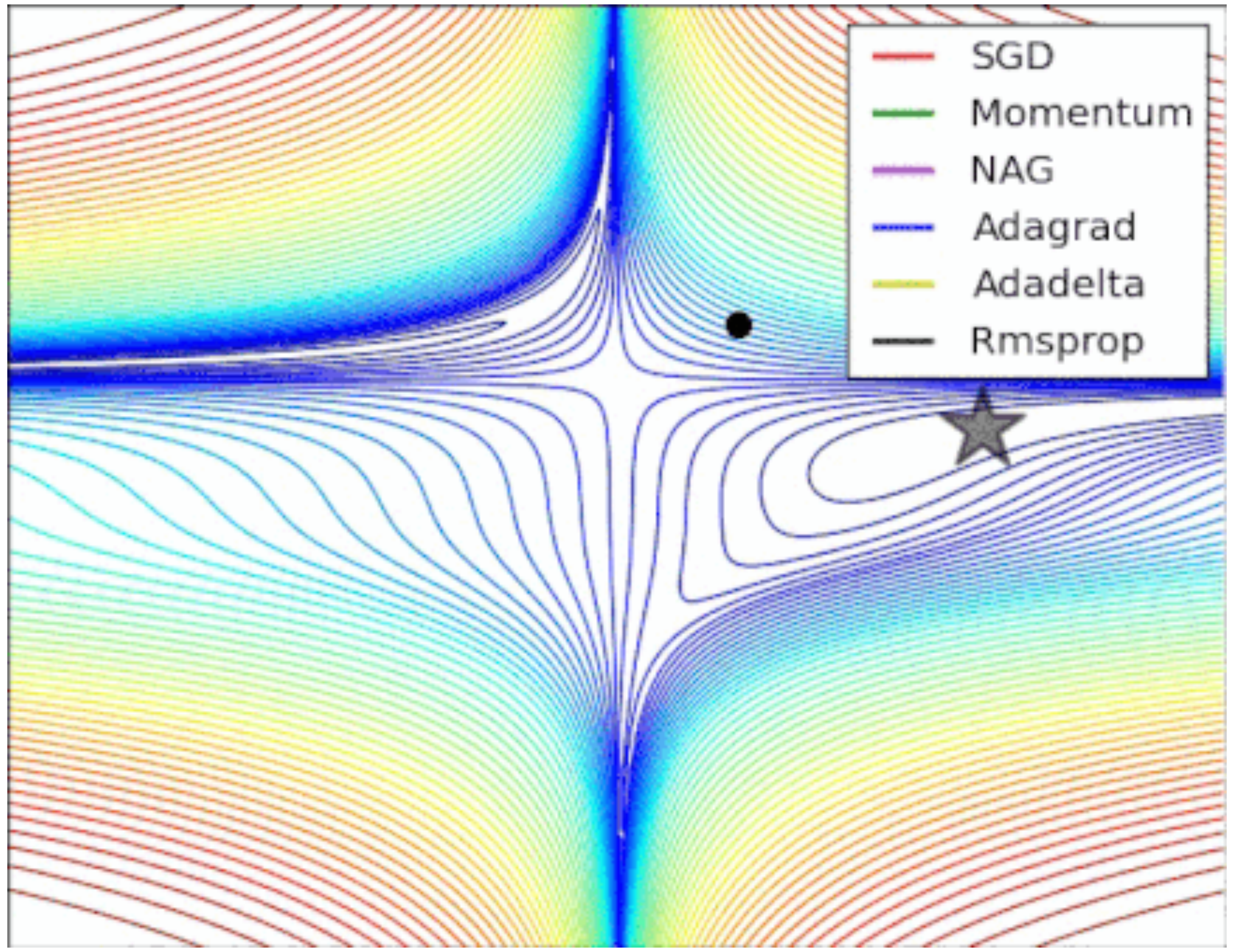


Summary of Optimizers

	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 Initializers



https://twitter.com/AlecRad

https://imgur.com/a/Hqolp

Learning Rate Schedules

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

From LMS, we know that large learning rate implies faster convergences, but more “misadjustment error” (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 tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler

```
# This function keeps the learning rate at 0.001 for the first ten epochs
# and decreases it exponentially after that.
def scheduler(epoch):
  if epoch < 10:
    return 0.001
  else:
    return 0.001 * tf.math.exp(0.1 * (10 - epoch))

callback = tf.keras.callbacks.LearningRateScheduler(scheduler)
model.fit(data, labels, epochs=100, callbacks=[callback],
          validation_data=(val_data, val_labels))
```

LearningRateScheduler() is a **callback** class built-in for you

you just need to pass it a schedule which returns eta as a function i in {0,1,2…} (epoch)

From LMS, we know that large learning rate implies faster convergences, but more “misadjustment error” (gradient noise)

Aside: Callbacks in tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks

Classes

`class BaseLogger` : Callback that accumulates epoch averages of metrics.

`class CSVLogger` : Callback that streams epoch results to a csv file.

`class Callback` : Abstract base class used to build new callbacks.

`class EarlyStopping` : Stop training when a monitored quantity has stopped improving.

`class History` : Callback that records events into a `History` object.

`class LambdaCallback` : Callback for creating simple, custom callbacks on-the-fly.

`class LearningRateScheduler` : Learning rate scheduler.

`class ModelCheckpoint` : Save the model after every epoch.

`class ProgbarLogger` : Callback that prints metrics to stdout.

`class ReduceLROnPlateau` : Reduce learning rate when a metric has stopped improving.

`class RemoteMonitor` : Callback used to stream events to a server.

`class TensorBoard` : Enable visualizations for TensorBoard.

`class TerminateOnNaN` : Callback that terminates training when a NaN loss is encountered.

These are built-in callbacks you can use

You can create your own custom callback
by building on this base class
(more details in discussion)

Common LR Schedules

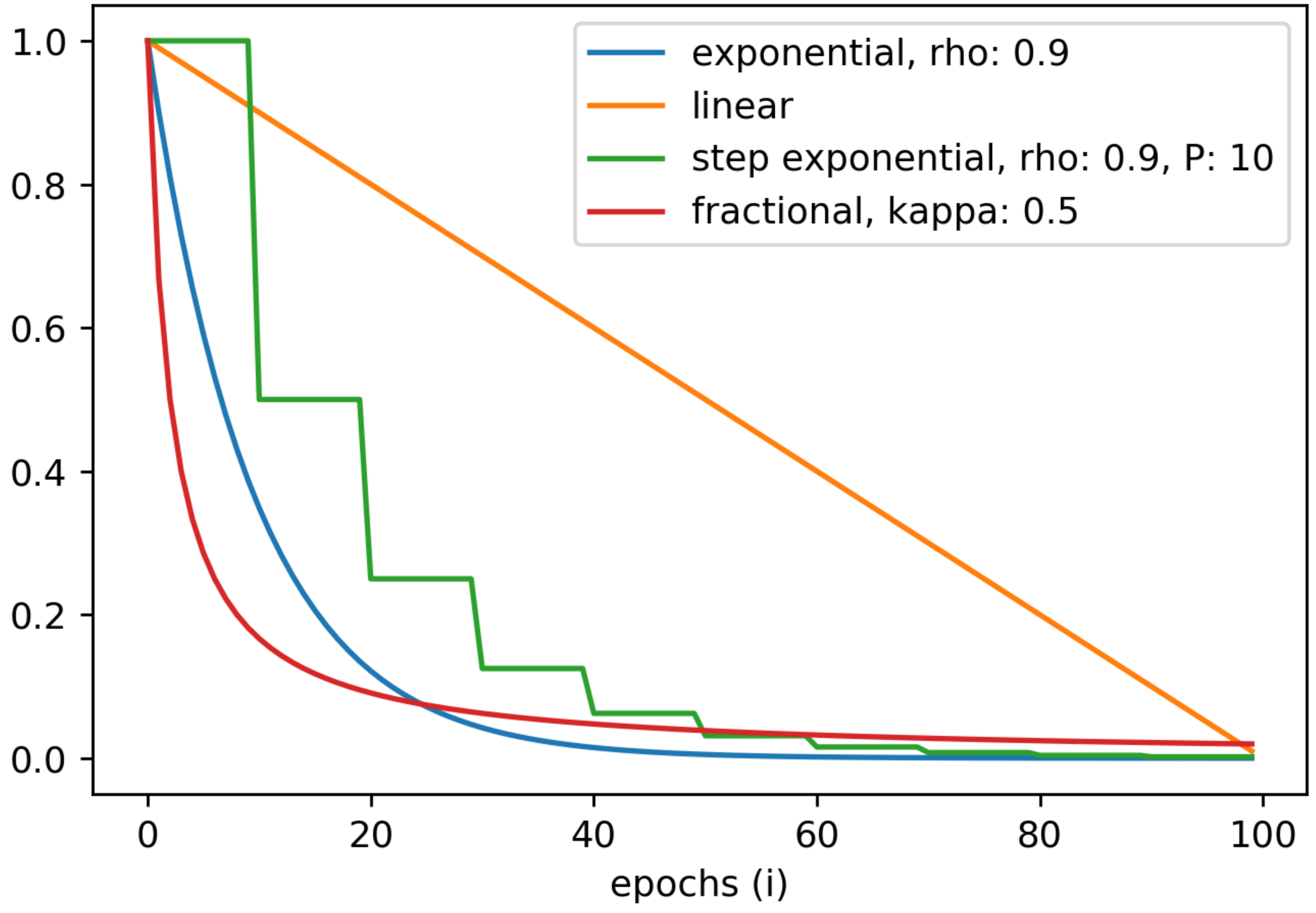
$\eta_i = \rho \eta_0$ Exponential Decay

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

$\eta_i = \eta_0 \rho^{\lfloor \frac{i}{P} \rfloor}$ Step Exponential Decay

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

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



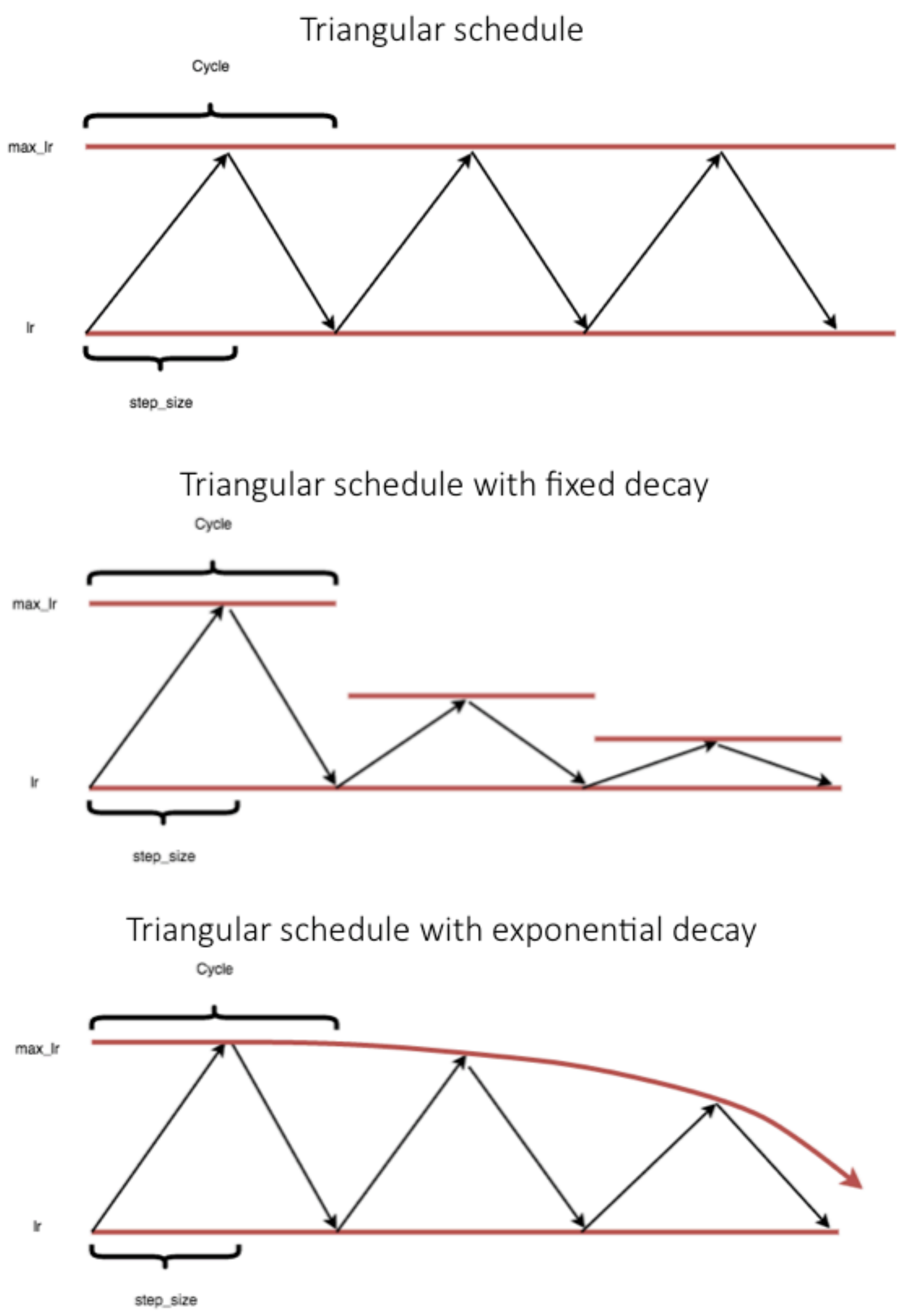
Another common LR schedule is to decrease the LR at specific epochs in a stepwise manner

e.g., at 50% and 75% of the total number of epochs: LR <— LR * 0.2

contributions from Sourya Dey

More Exotic LR Schedules

Triangular Schedules



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

Cosine Schedules

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

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

cosine schedule is "experimental" in tf.keras

https://www.tensorflow.org/api_docs/python/tf/keras/experimental/CosineDecay

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

contributions from Sourya Dey

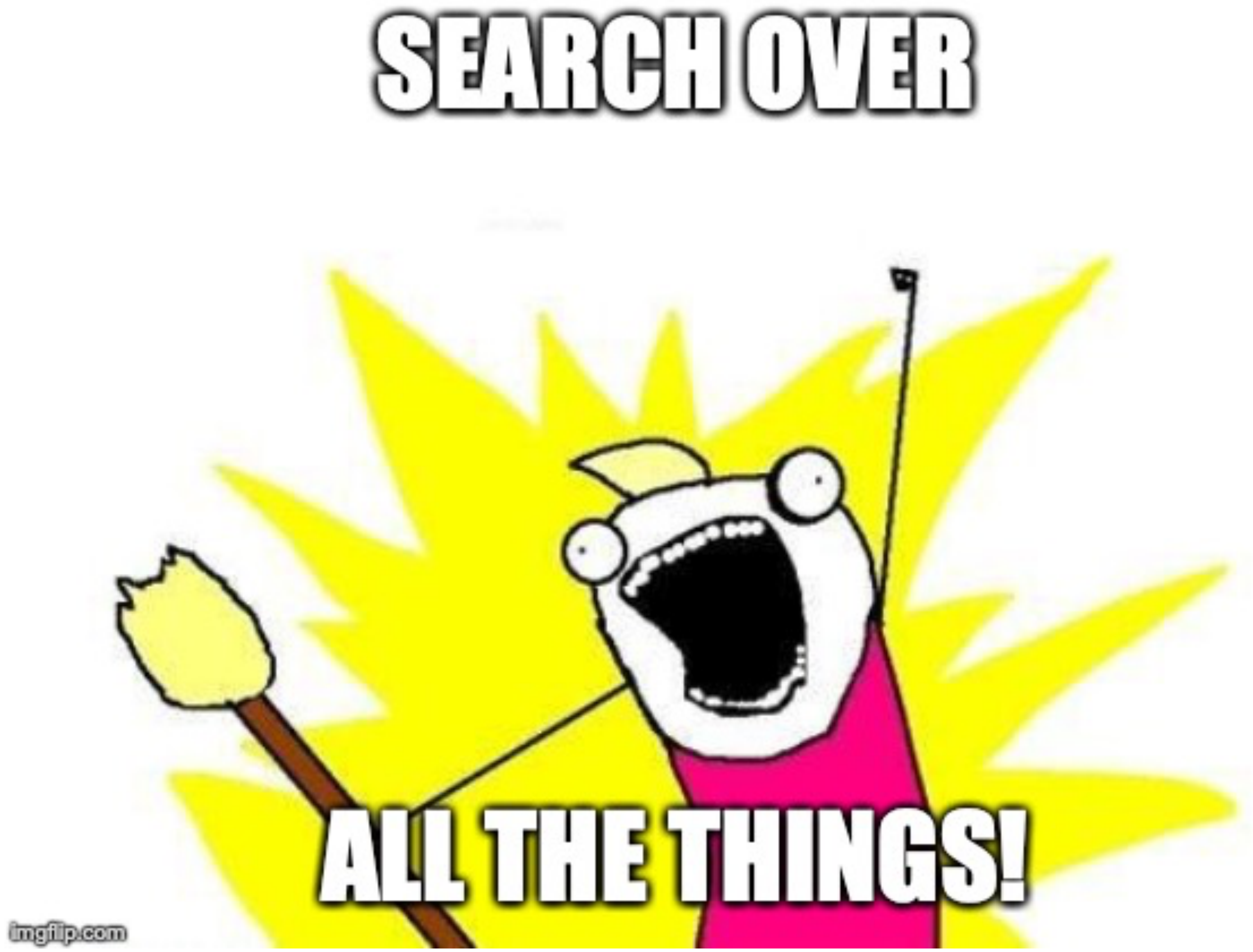
Outline for Slides

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

Is this 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



Some Big-Picture 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 some 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

use the ReduceLROnPlateau() callback as a start to LR scheduling or simple step LR schedules

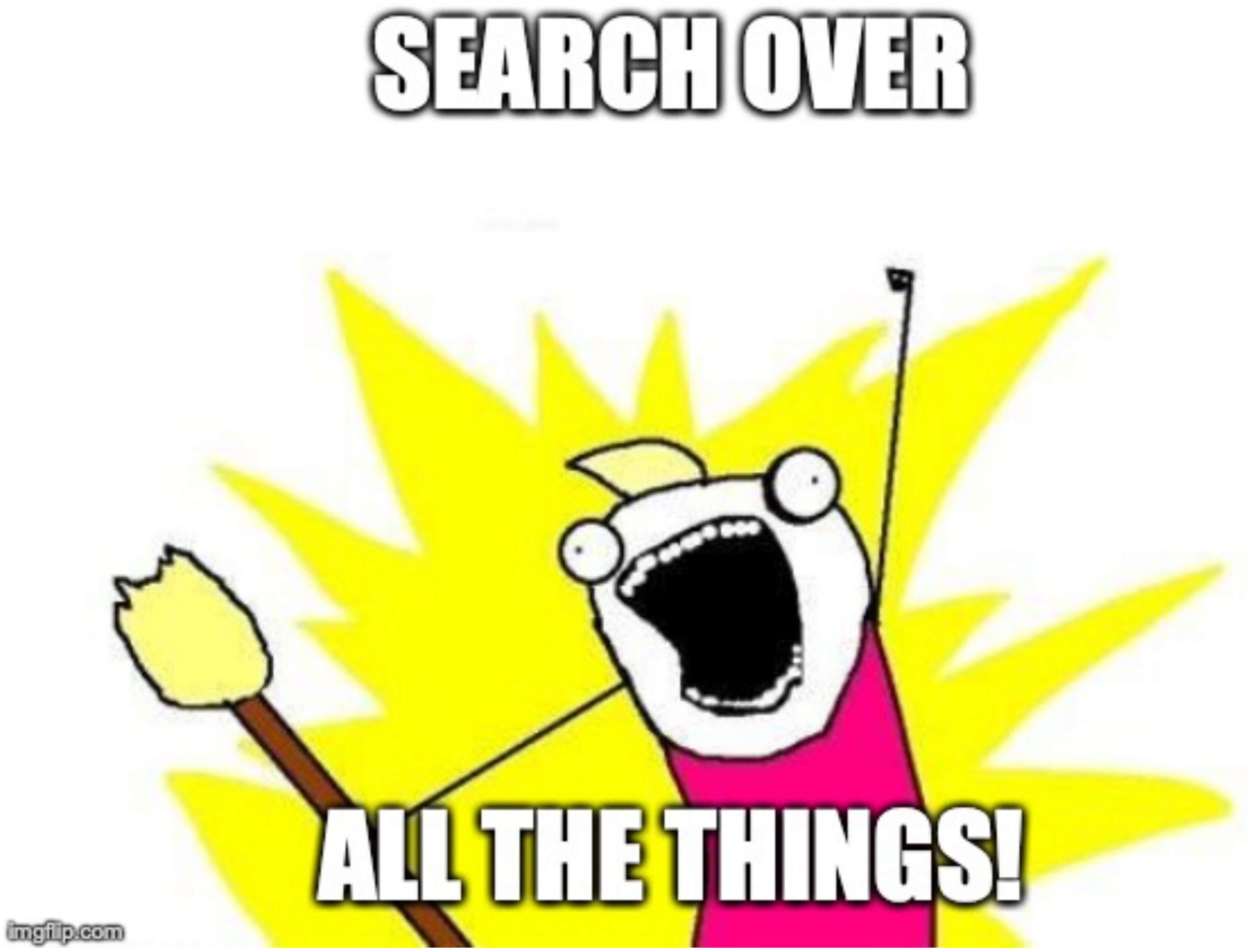
A lot of focus on this in the literature, but designing your dataset is more important (this is fine tuning for real world applications IMO)

Automated Network Architecture Search and Hyperparameter Optimization

We will have a guest lecture by Sourya Dey on this research topic

Sourya is a current PhD student

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



Outline for Slides

- Universal Approximation Theorem
    - Why Deep?
- A Gentle Introduction to tensorflow.keras
- Vanishing gradient and activations
- Weight initialization
- Cost functions, regularization, dropout
- Optimizers
- 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.

gamma and **beta** are trainable parameters

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

During inference, replace the min-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).