USC Viterbi School of Engineering

TRAINING DEEP NEURAL NETWORKS I

EE 541 – UNIT 6A

DR. BRANDON FRANZKE

Spring 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

UNIVERSAL APPROXIMATION THEOREM

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 $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define

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

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

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

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

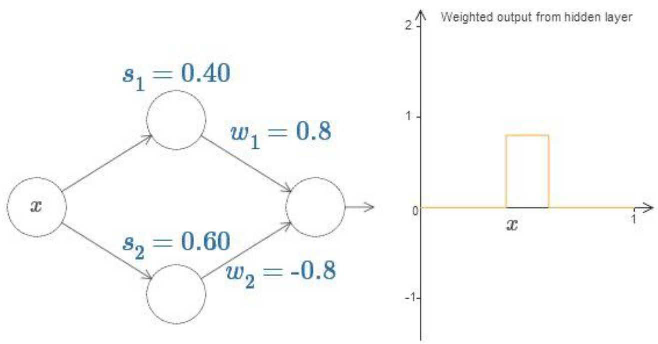
A single hidden layer MLP with squashing activation in the hidden layer and linear output layer can approximate any “engineering function”

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

UNIVERSAL APPROXIMATION THEOREM

How does the intuition behind this work?

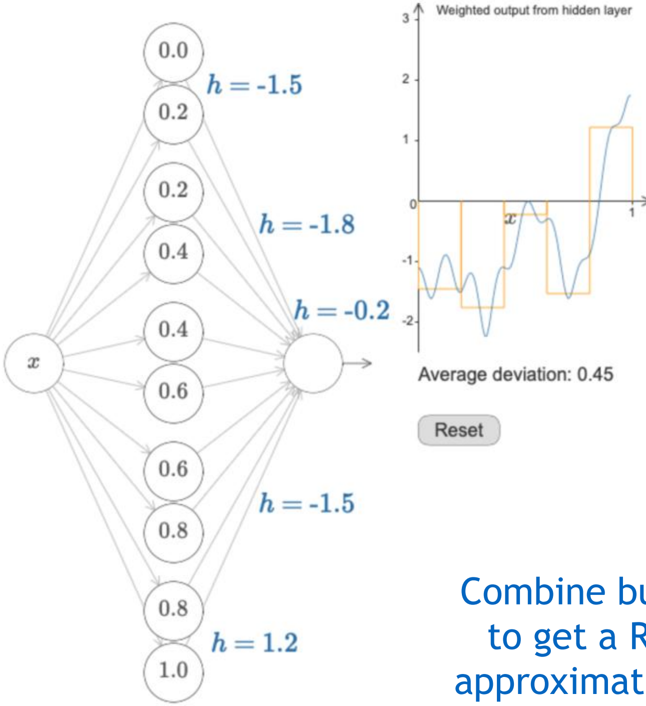
http://neuralnetworksanddeeplearning.com/chap4.html



can create a "bump" function

done by choosing large weights in layer 1

s = -b/w (step position)

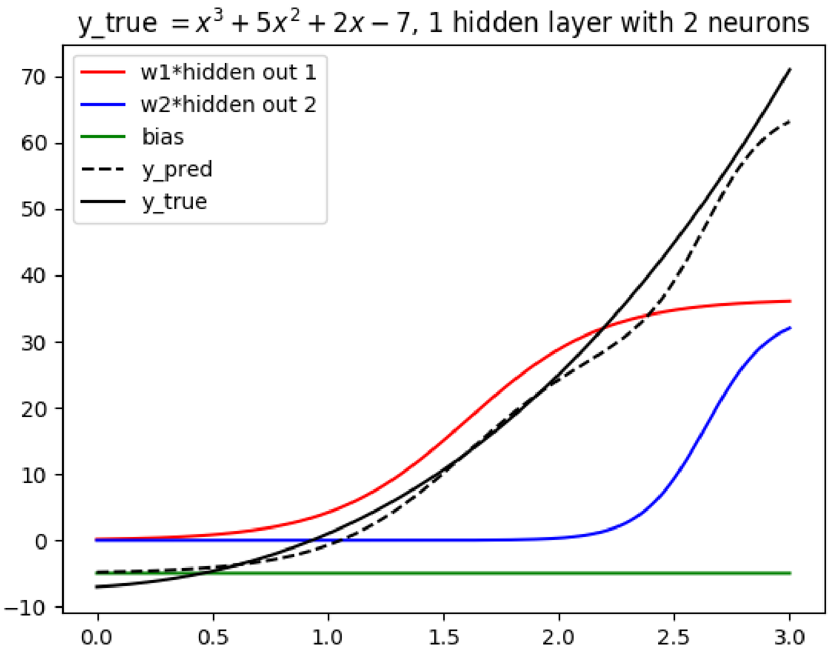


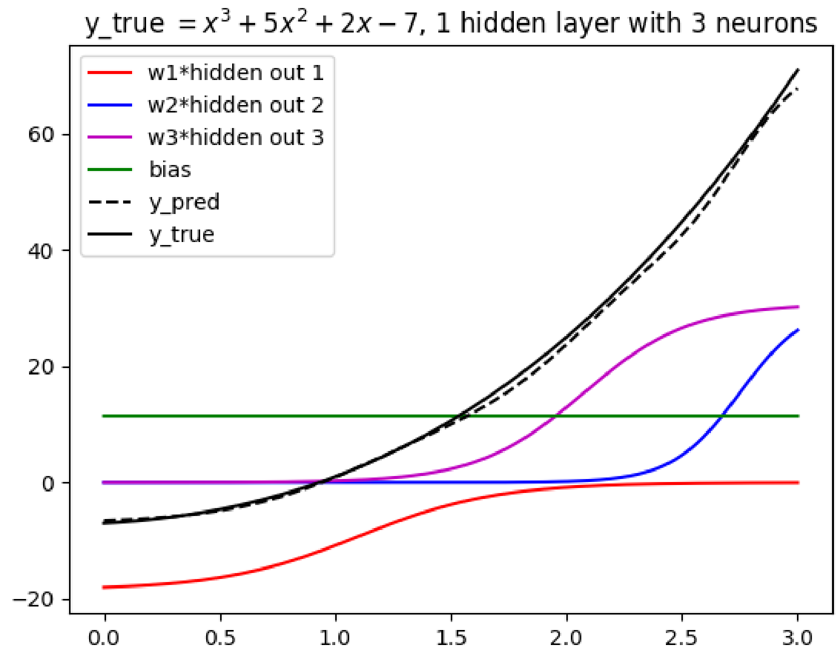
Combine bump functions to get a Riemann-like approximation with many nodes in hidden layer

UNIVERSAL APPROXIMATION THEOREM

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

http://neuralnetworksanddeeplearning.com/chap4.html



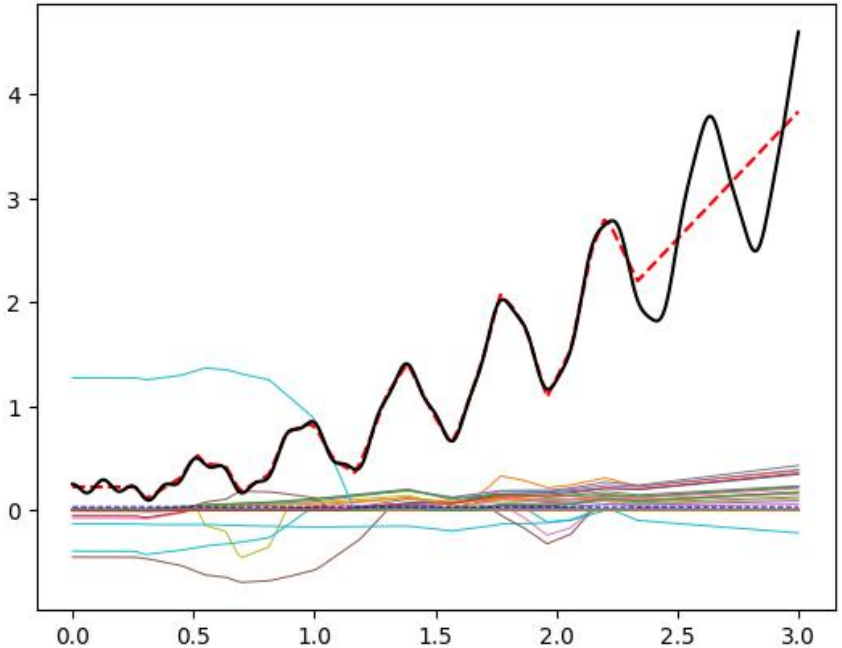


UNIVERSAL APPROXIMATION THEOREM

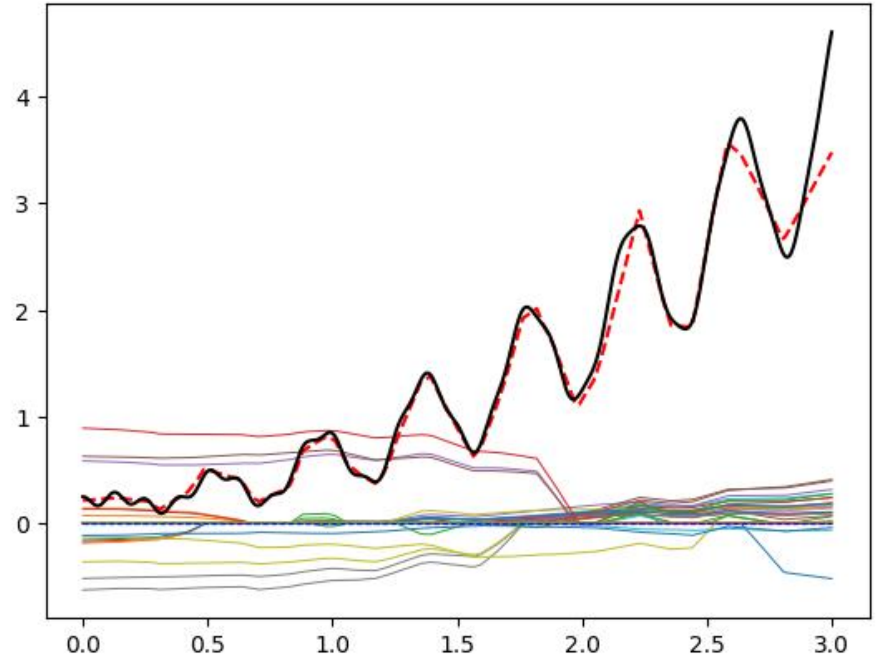
What happens when we train a neural net on Neilson's example?

```python
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 can learn this good approximation

3) There are inherent advantages to more hidden layers

multiple layers can learn stages of classification or “case switches”

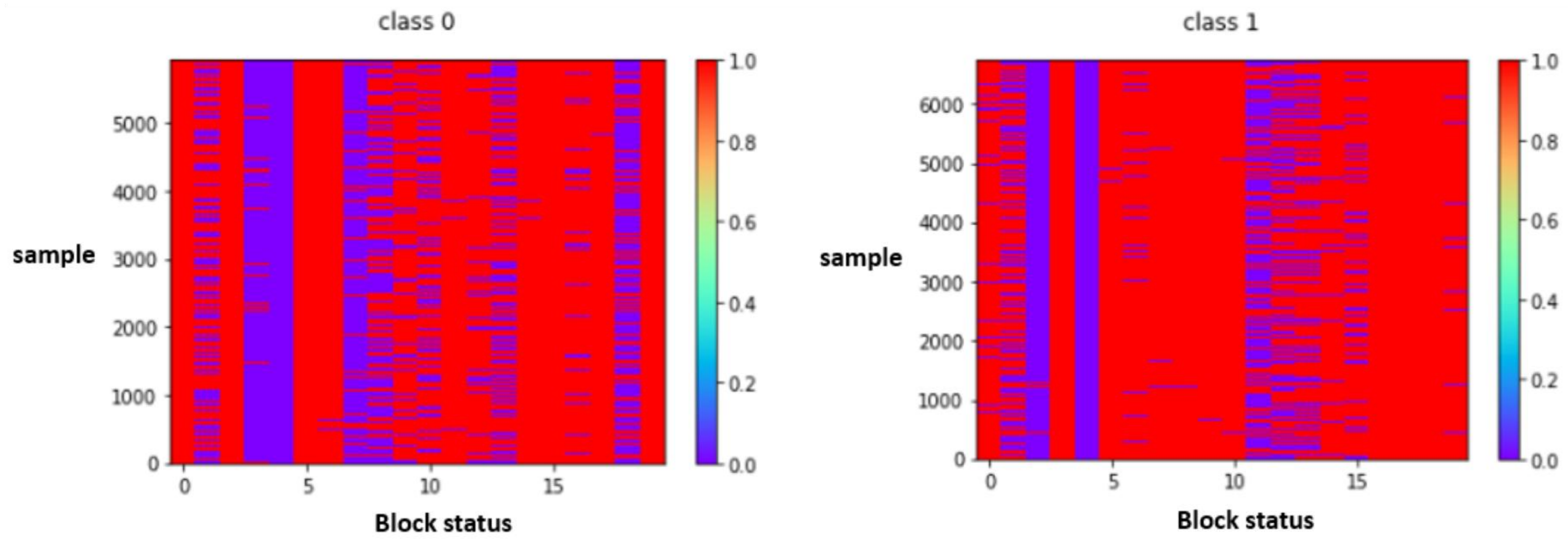
e.g.,

Layer 1: 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 -- *e.g.*, “clamps/conditionals” --- multiple layers can model this more explicitly

EXAMPLE FROM CLASS PROJECT



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

Intuition: ReLU-based MLP is *toggling switches* based on classification.
Then applying a linear mapping (these are like the clamps/conditionals)

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?

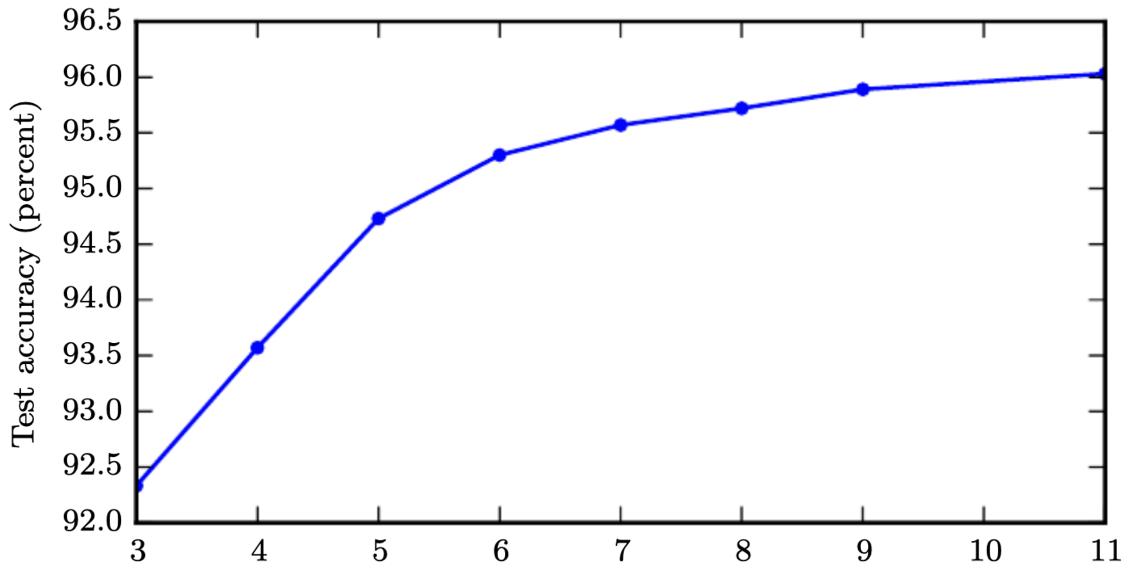


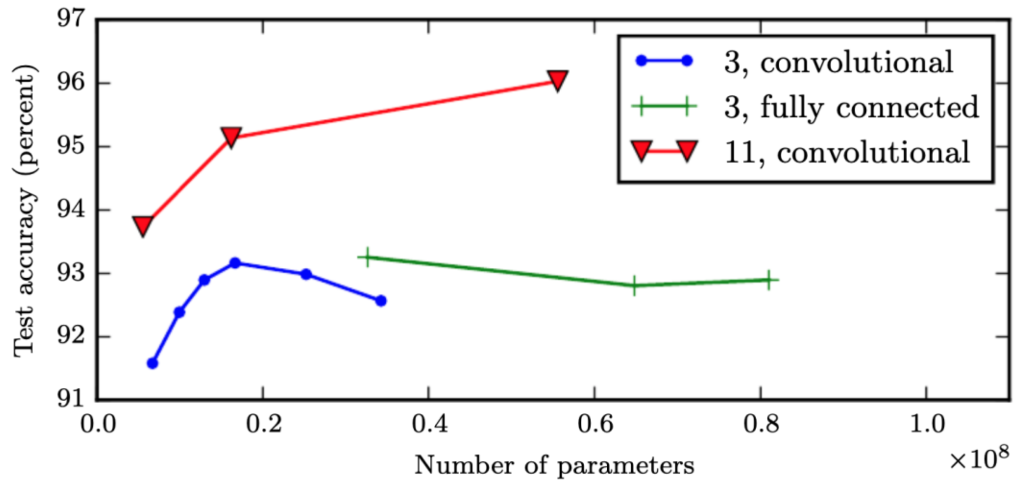
Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control 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?



deeper models tend to perform better

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).

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

USC Viterbi School of Engineering

PYTORCH

GENTLE INTRODUCTION TO PYTORCH

Use PyTorch 2.4 or 2.5

You can install PyTorch through anaconda

best to set up virtual-environment with `conda`

(or use pyenv to create and manage minimal virtualenvs)

I use: conda, **PyTorch 2.4 (upg: 2.5)**, MacOS 15, **Python 3.11**
TorchVision 0.15, TensorBoard 2.17 (on PC: CUDA via conda)

CONDA ENVIRONMENT

```
conda create --name ee541_work python=3.11
conda activate ee541_work

# mac (mps)
#conda install pytorch::pytorch torchvision torchaudio -c pytorch
# pc (with cuda), check cuda version e.g. > 11.7 / 11.8
#conda install pytorch torchvision torchaudio pytorch-cuda -c pytorch -c
nvidia

conda install numpy scipy opencv matplotlib
conda install h5py jupyter tensorboard seaborn tqdm pandas
conda install -c conda-forge torchinfo
# extra
pip install torchviz
```

INTRODUCTION TO PYTORCH

Let's code our first neural network!

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

train_set = torchvision.datasets.FashionMNIST(root = "./data", train = True, download = True,
                                              transform = transforms.ToTensor())
test_set = torchvision.datasets.FashionMNIST(root = "./data", train = False, download = True,
                                             transform = transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(train_set, batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=100, shuffle=False)

model = torch.nn.Sequential(
        nn.Linear(in_features=28*28, out_features=128),
        nn.ReLU(),
        nn.Linear(in_features=128, out_features=10)
        #nn.Softmax(dim=1)
)

loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

num_epochs =
for epoch in range(num_epochs):
    [...]
```

typical imports

PyTorch has standard datasets built-in – auto-download

loaders feed data to your model

define a **<u>sequential</u> network**

specify loss function and optimizer

and training loop… details shortly

INTRODUCTION TO PYTORCH

Same implementation but use the "Functional API" to define the model

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

train_set = torchvision.datasets.FashionMNIST(root = "./data", train = True, download = True,
                                              transform = transforms.ToTensor())
test_set = torchvision.datasets.FashionMNIST(root = "./data", train = False, download = True,
                                             transform = transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(train_set, batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=100, shuffle=False)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden = nn.Linear(num_pixels, 128)
        self.output = nn.Linear(128, 10)

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

model = Net()

loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

num_epochs =
for epoch in range(num_epochs):
    [...]
```

define a torch.nn network (modular)

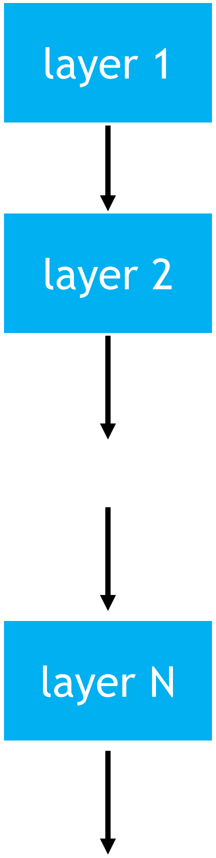
PYTORCH — DEFINING THE MODEL

Sequential

simple, quick

not very flexible

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



Use the `nn.Module` object for most models.

(next slide)

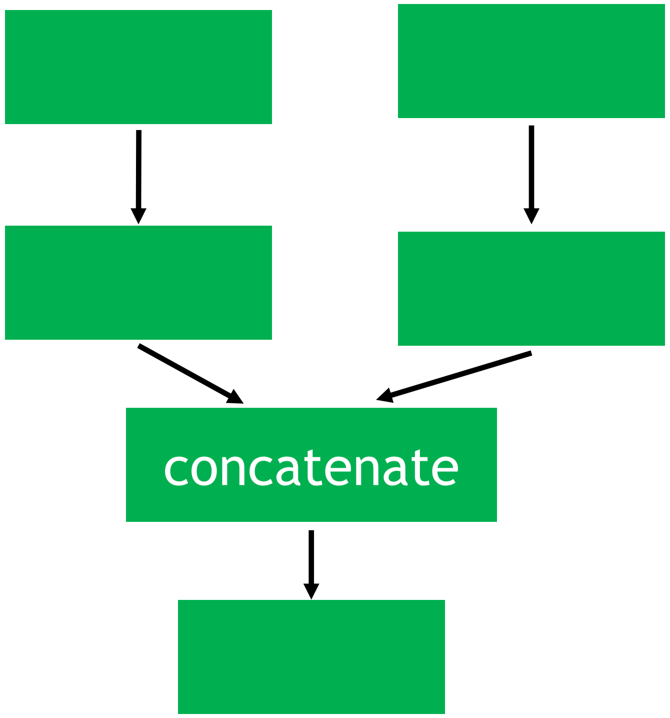
PYTORCH — DEFINING THE MODEL

Modular -- `torch.nn.Module`

little more setup (net + forward)

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



PYTORCH— VIEWING MODEL STRUCTURE

```
print(model)
```

```
Net(
  (hidden): Linear(in_features=784, out_features=128, bias=True)
  (output): Linear(in_features=128, out_features=10, bias=True)
)
```

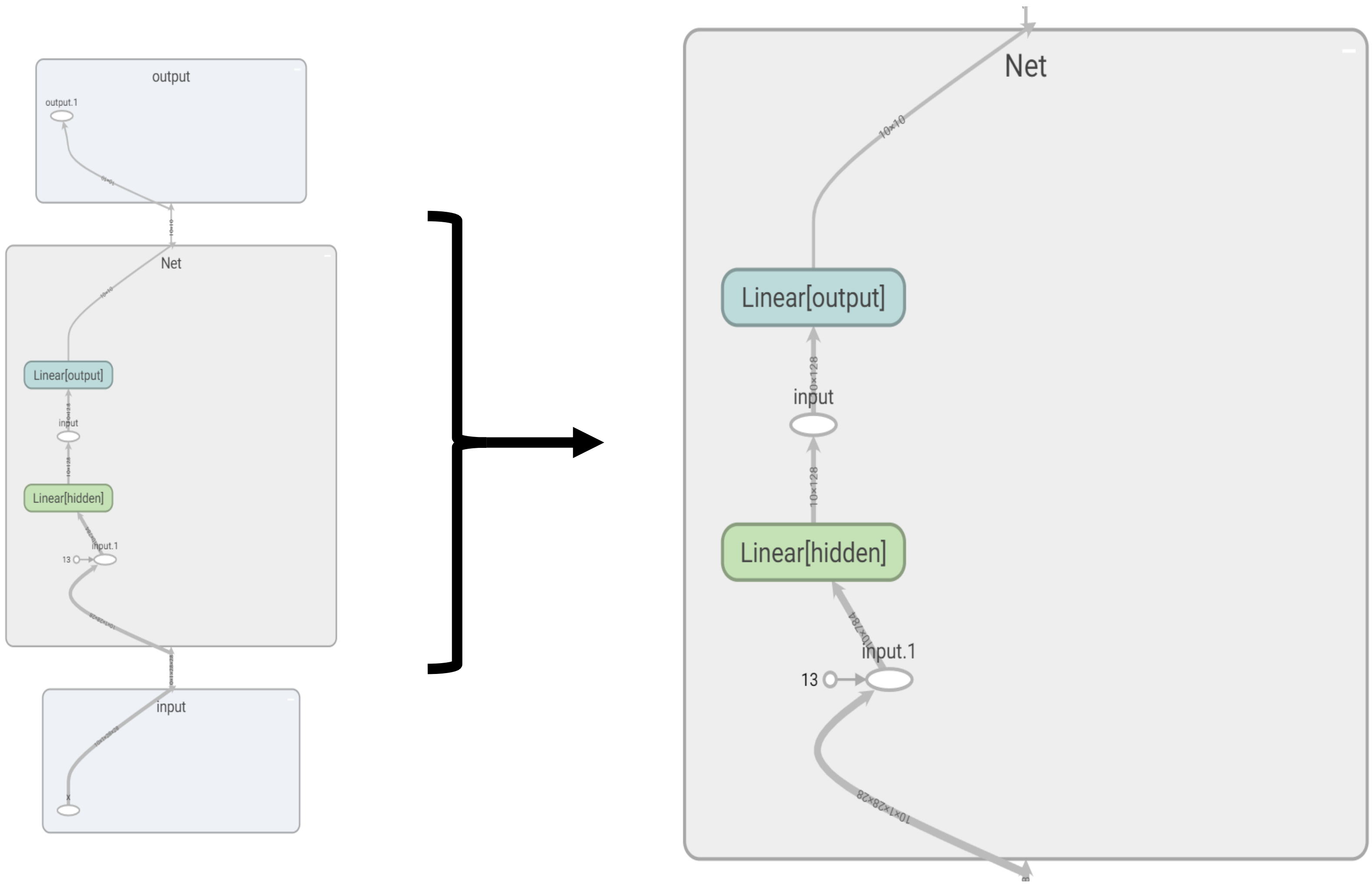
leaves something to be desired, so…

```

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
            Linear-2              [-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
----------------------------------------------------------------
```

even better: TensorBoard – GUI + inspection + Evaluation

TENSORBOARD — INSPECT INTERNAL MODEL STRUCTURE



PYTORCH — TRAINING LOOP (NO VALIDATION)

```python
num_epochs = 4
count = 0
for epoch in range(num_epochs):
    correct = 0
    for images, labels in train_loader:
        count += 1
        input = images.view(-1, 28*28)

        # forward pass
        outputs = model(input)
        loss = loss_func(outputs)

        # backprop
        optimizer.zero_grad()
        loss.backward()

        # optimize
        optimizer.step()

        # not normally in training
        predictions = torch.max(outputs, 1)[1]
        correct += (predictions == labels).sum().numpy()

    print(f'Epoch: {epoch+1:02d}, Iteration: {count:5d}, Loss: {loss.data:.4f}, ' +
          f'Accuracy: {100 * correct/len(train_loader.dataset):2.3f}%')

print('Finished Training')
```

```
Epoch: 01, Iteration:   600, Loss: 0.7919, Accuracy: 69.465%
Epoch: 02, Iteration:  1200, Loss: 0.5634, Accuracy: 80.110%
Epoch: 03, Iteration:  1800, Loss: 0.2925, Accuracy: 82.467%
Epoch: 04, Iteration:  2400, Loss: 0.4984, Accuracy: 83.615%
Finished Training
```

PYTORCH — MONITORING PERFORMANCE (TRAINING)

Gather loss and accuracy. Plot later.

```
# for plots
loss_list = []
iteration_list = []
accuracy_list = []

num_epochs = 4
count = 0
for epoch in range(num_epochs):
    correct = 0
    for images, labels in train_loader:
        count += 1
        input = images.view(-1, 28*28)

        # forward pass
        outputs = model(input)
        loss = loss_func(outputs)

        # backprop
        optimizer.zero_grad()
        loss.backward()

        # optimize
        optimizer.step()

        # not normally in training
        predictions = torch.max(outputs, 1)[1]
        correct += (predictions == labels).sum().numpy()

        loss_list.append(loss.data)
        iteration_list.append(count)
        accuracy_list.append(correct / len(images))

    print(f'Epoch: {epoch+1:02d}, Iteration: {count:5d}, Loss: {loss.data:.4f}, ' +
          f'Accuracy: {100 * correct/len(train_loader.dataset):2.3f}%')

print('Finished Training')
```

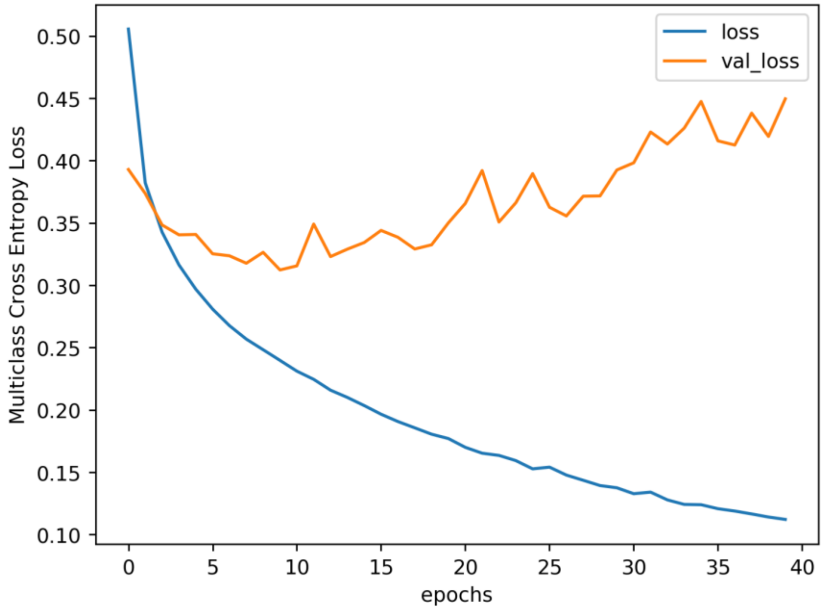
Capture at desired interval (per batch here)

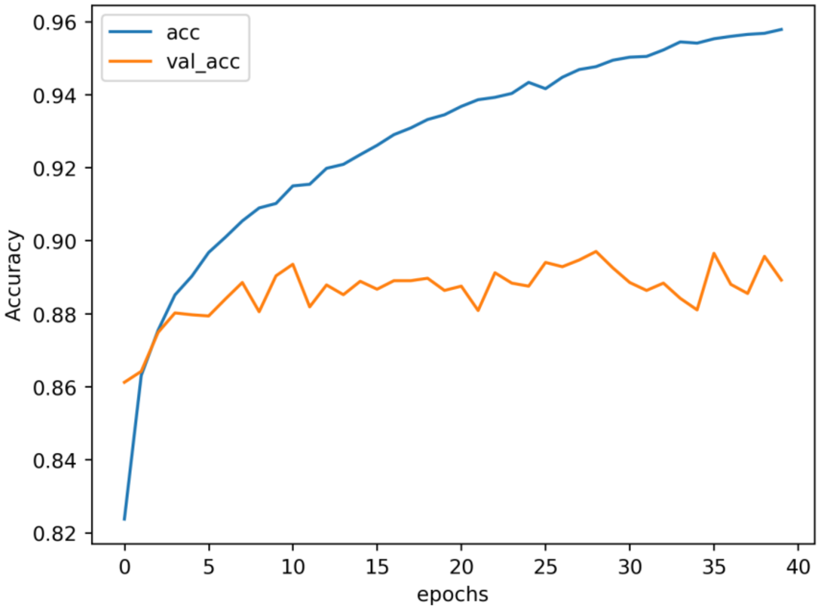
then use standard plotting tools

```
fig = plt.figure()
ax = fig.gca()

fig.add_subplot(1, 2, 1)
plt.plot(iteration_list, loss_list)

ax.set_xlabel('Iteration')
ax.set_ylabel('Loss')

fig.add_subplot(1, 2, 2)
plt.plot(iteration_list, accuracy_list)

ax.set_xlabel('Iteration')
ax.set_ylabel('Accuracy')

plt.show()
```

PYTORCH — CHECK PERFORMANCE

results of our training run…





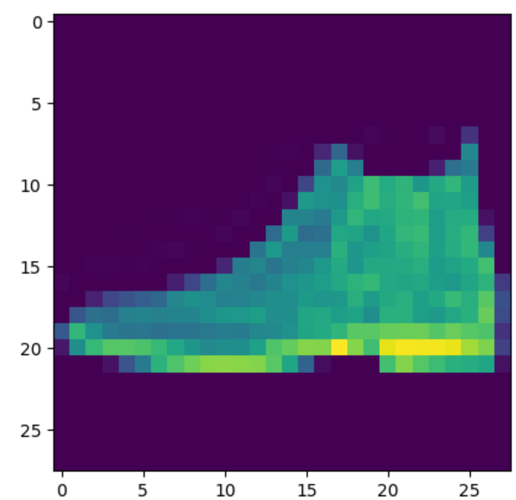
over-fitting (bad!)

PYTORCH — CHECK 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)

PYTORCH — CHECK PERFORMANCE

manual inference on a single image

```
image, label = test_set[0]
prediction = model(image.view(-1, 28*28)).reshape(10)
class_decision = np.argmax(prediction)
for m,label in enumerate(classes):
    if m == class_decision:
        print(f'class={label:15s} soft-decision: {prediction[m]:>9.5f} (hard decision)')
    else:
        print(f'class={label:15s} soft-decision: {prediction[m]:>9.5f}')
```

```
class=T-shirt/top     soft-decision:  -0.09017
class=Trouser         soft-decision:  -0.06262
class=Pullover        soft-decision:  -0.04838
class=Dress           soft-decision:   0.04331
class=Coat            soft-decision:  -0.11377
class=Sandal          soft-decision:   0.03234
class=Shirt           soft-decision:  -0.04101
class=Sneaker         soft-decision:   0.06401
class=Bag             soft-decision:  -0.02207
class=Ankle Boot      soft-decision:   0.07601 (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

Pass many images to model() and the output represents batch predictions

PYTORCH — TEST SET PERFORMANCE

```python
    with torch.no_grad():
        total = 0
        correct = 0

        for images, labels in test_loader:
            model.eval()
            images = images.to(device)

            test = images.view(-1, num_pixels)
            outputs = model(test).cpu()

            predictions = torch.max(outputs, 1)[1]
            correct += (predictions == labels).sum().numpy()
            total += len(labels)

        accuracy = correct * 100 / total

        loss_list.append(loss.data)
        iteration_list.append(count)
        accuracy_list.append(accuracy)

        print(f'Epoch: {epoch+1:02d}, Iteration: {count:5d}, Loss: {loss.data:.4f}, Accuracy: {accuracy:.3f}%')
```

After 4 epochs

(2400 iterations)

Loss: **0.2738**

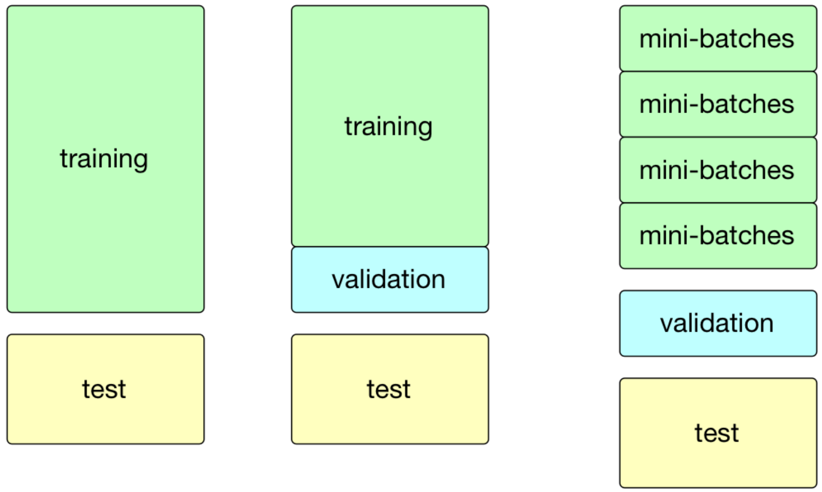
Accuracy: **86.65%**

USC Viterbi School of Engineering

DEALING WITH DATA

TRAIN VS. TEST

TRAINING (+ VAL)/TEST SPLIT



weights, biases

(training-)training: use this for SGD learning — trainable parameters

(training-)validation: use this for SGD learning — hyper-parameters

test: only use this when you are done to verify

learning rate, batch size, etc.

TRAINING (+VAL)/TEST SPLIT

Typical Train/Validation/Test split:

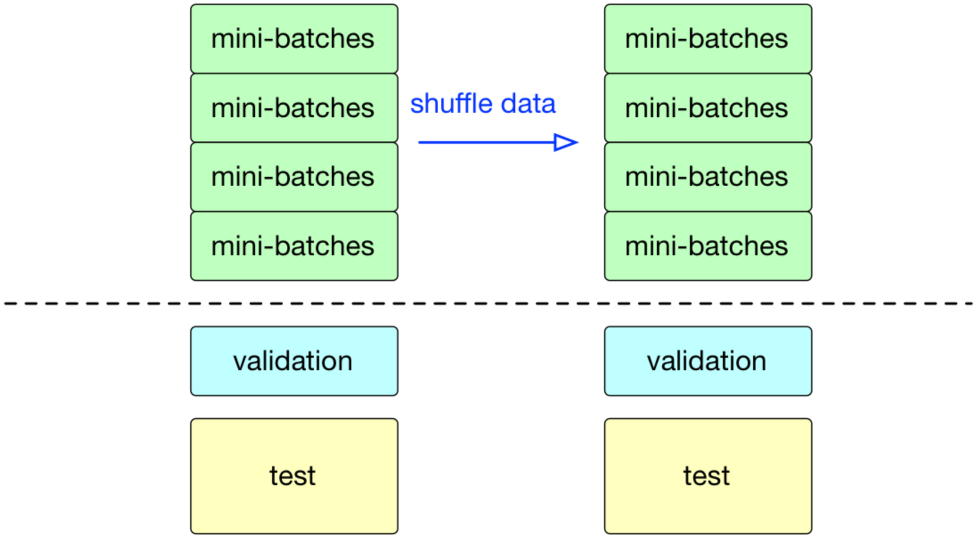
70/15/15

val and **test** splits must be large enough to capture natural variation in the data

val and **test** splits must be large enough to allow reliable classification error estimates

So: want lots of data

TRAINING (+ VAL)/TEST SPLIT



shuffle all data in training (not including validation) after each epoch

```
perm = np.random.permutation(N_train)
x_train = x_train[perm]
y_train = y_train[perm]
```

good practice to shuffle all of the data once before the train/val/test split

TRAINING (+ VAL)/TEST SPLIT

(mini)-batch: do one SGD update (averaging) per mini-batch

(mini)-batch size: number of data examples per mini-batch

epoch: one training run through all of the training data

iteration: number of mini-batches per epoch

typically, we “test” the model on the validation data at the end of each epoch

TRAINING (+ VAL)/TEST SPLIT

Example

100,000 total (x_n, y_n) (shuffle it all once)

70,000 train
15,000 val
15,000 test

Suppose: batch size = 70:

1000 mini-batches in the training data (1000 iterations per epoch)

1000 gradient updates in an epoch, each averaged over 70 samples

these are typically processed serially: batch 1, batch 2, etc.

gradient updates are serial

(can change with many parallel compute nodes)

run inference (forward only) on val data after each epoch

monitor learning curve, iteration hyper-hyper-parameter search…

when done, run on test

USC Viterbi School of Engineering

VANISHING GRADIENT

VANISHING GRADIENT PROBLEM

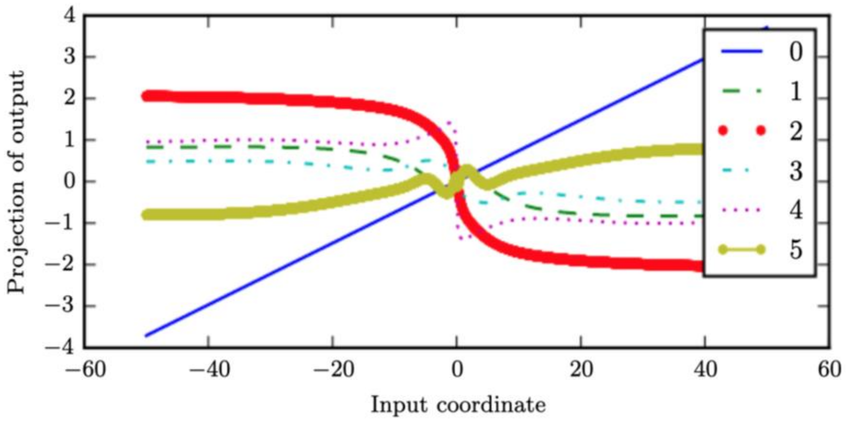


Figure 10.15: Repeated function composition. When composing many nonlinear functions (like the linear-tanh layer shown here), the result is highly nonlinear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many alternations between increasing 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

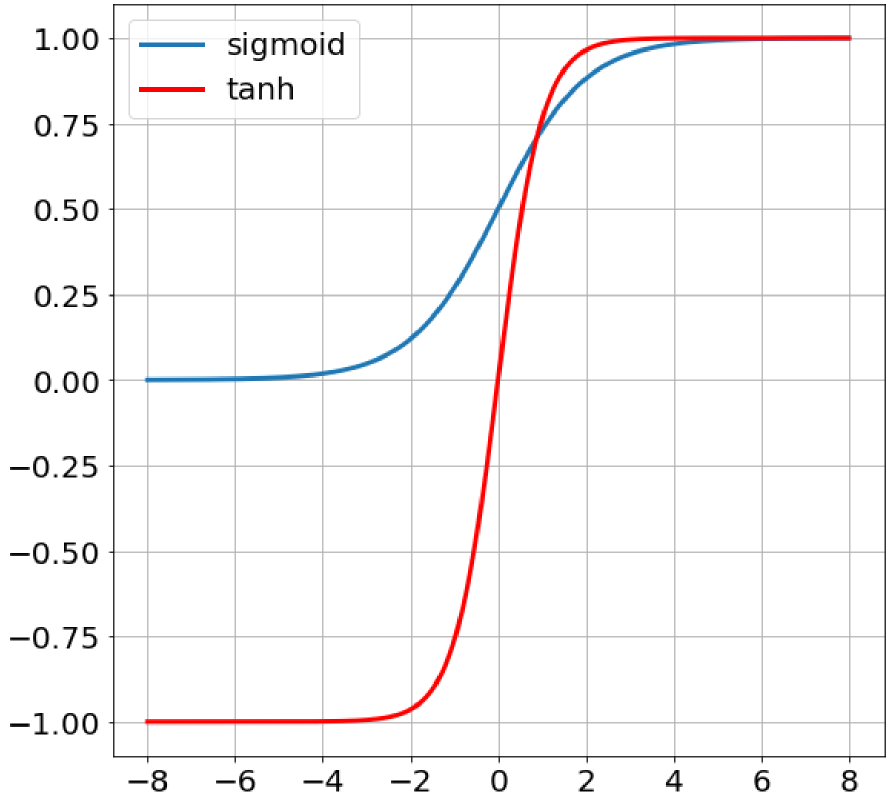
due to the squashing activation compounded effects

See section 10.7 of Deep Learning book for further discussion

VANISHING GRADIENT PROBLEM
SQUASHING ACTIVATIONS

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

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



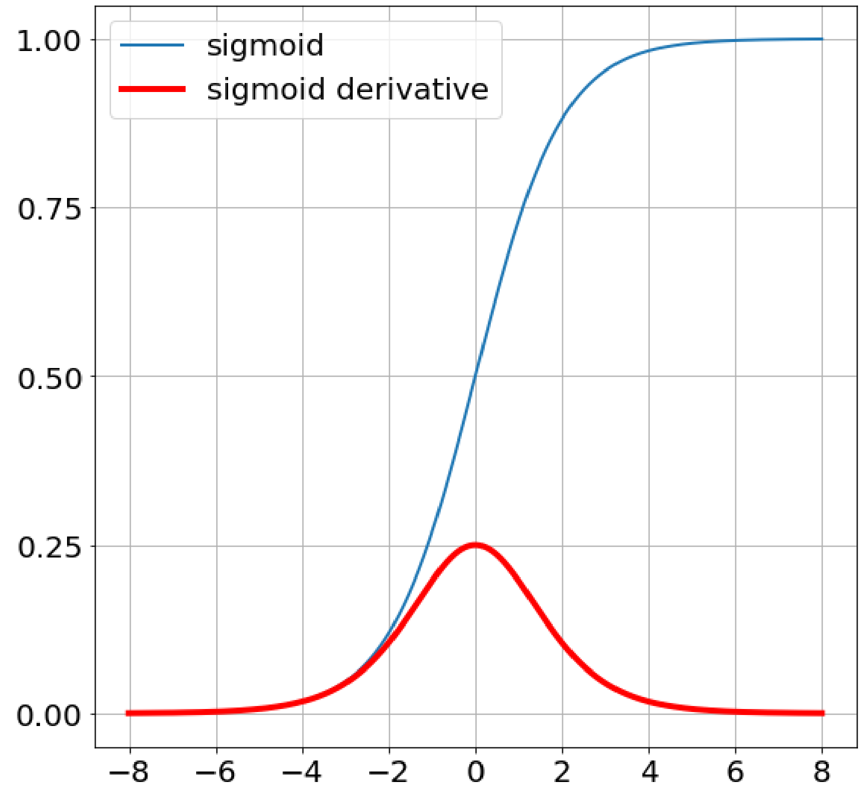
the gradient can get small as we back-prop

due to the squashing activation compounded effects

VANISHING GRADIENT PROBLEM
SQUASHING ACTIVATIONS

$$\sigma'(x) = \sigma(x)\left(1 - \sigma(x)\right)$$

the maximum value of $\dot{\sigma}(\cdot)$ is 0.25...



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

VANISHING GRADIENT PROBLEM - RELU ACTIVATIONS

Biologically inspired - neurons firing vs not firing

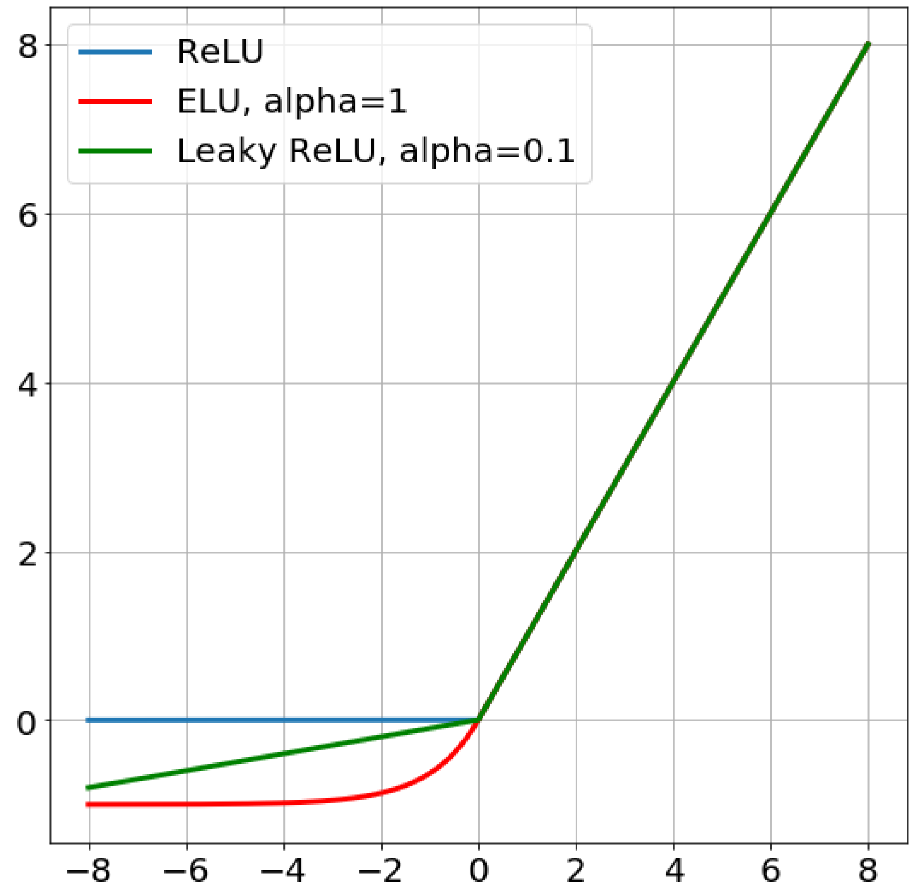
Solves vanishing gradient problem

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

ACTIVATIONS IN PYTORCH

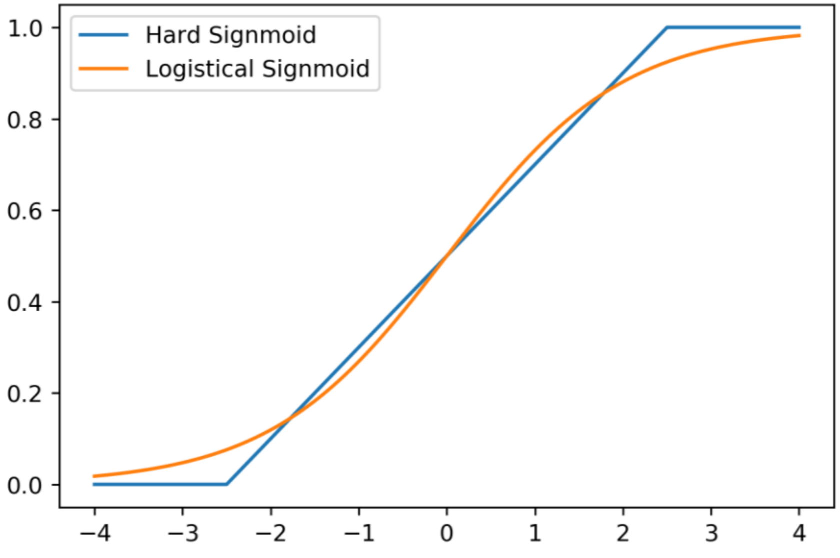
Non-linear Activations (weighted sum, nonlinearity)

`nn.ELU`	Applies the element-wise function:
`nn.Hardshrink`	Applies the hard shrinkage function element-wise:
`nn.Hardsigmoid`	Applies the element-wise function:
`nn.Hardtanh`	Applies the HardTanh function element-wise
`nn.Hardswish`	Applies the hardswish function, element-wise, as described in the paper:
`nn.LeakyReLU`	Applies the element-wise function:
`nn.LogSigmoid`	Applies the element-wise function:
`nn.MultiheadAttention`	Allows the model to jointly attend to information from different representation subspaces.
`nn.PReLU`	Applies the element-wise function:
`nn.ReLU`	Applies the rectified linear unit function element-wise:

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

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden = nn.Linear(num_pixels, 128)
        self.output = nn.Linear(128, 10)

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

ACTIVATIONS IN PYTORCH



CLASS `torch.nn.Hardsigmoid` [SOURCE]

Applies the element-wise function:

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \le -3, \\ 1 & \text{if } x \ge +3, \\ x/6 + 1/2 & \text{otherwise} \end{cases}$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Hardsigmoid()
>>> input = torch.randn(2)
>>> output = m(input)
```

`hard_sigmoid` sometimes used to reduce computation

ACTIVATIONS IN PYTORCH

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

soft-max:
produces $M \times 1$ probability mass function use for M-ary classification between mutually exclusive classes (*i.e.*, "1-hot")

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

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

USC Viterbi
School of Engineering

PARAMETER INITIALIZATION

WEIGHT (AND BIAS) INITIALIZATION

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

how do we initialize θ?

empirical observation: some initializations are better than others

zero initialization?

all linear activations are 0...

the δ's will be 0 too...

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

try random initialization...?

WEIGHT (AND BIAS) INITIALIZATION

Xavier (Glorot) Normal Initialization

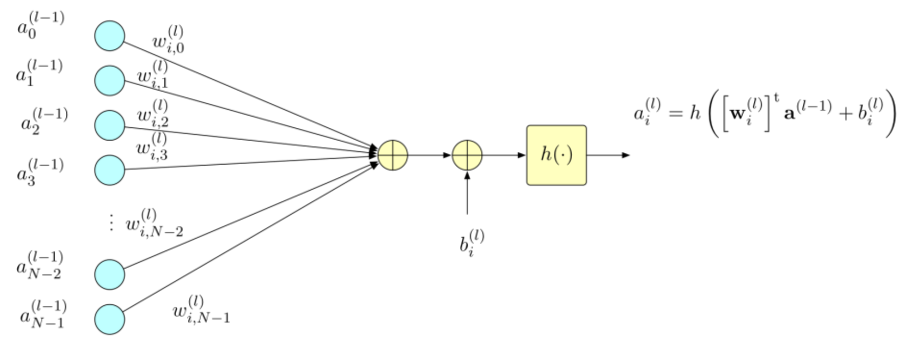
Consider a linear function:
(assume all w and x are I.I.D.)

$$y = w_1 x_1 + w_2 x_2 + \cdots + w_N x_N$$

$$\mathrm{Var}(y) = N \mathrm{Var}(w) \mathrm{Var}(x)$$

$$\text{if } \mathrm{Var}(w) = \frac{1}{N}$$

$$\text{then } \mathrm{Var}(y) = \mathrm{Var}(x)$$



This suggests:

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

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

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

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

WEIGHT (AND BIAS) INITIALIZATION

Xavier (Glorot) Uniform Initialization

use same second moments with uniform initialization….

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

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

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

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

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

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

WEIGHT (AND BIAS) INITIALIZATION

Kaiming Initialization

Xavier does not account for nonlinear activations $E[x_k^2] \neq 0$ ($e.g.$, ReLU)

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

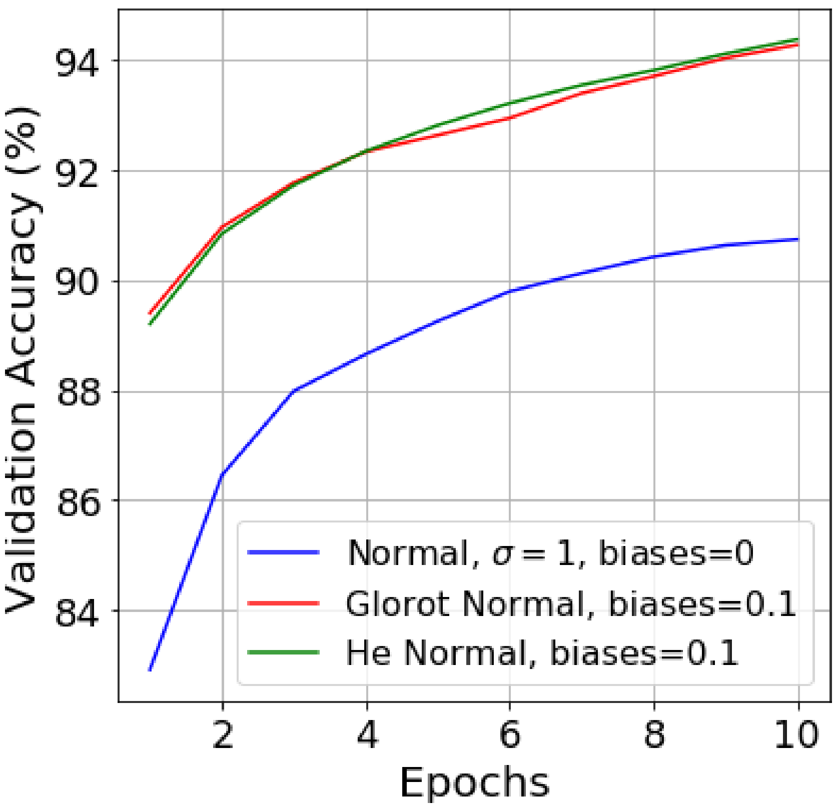
Kaiming Normal Initialization

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

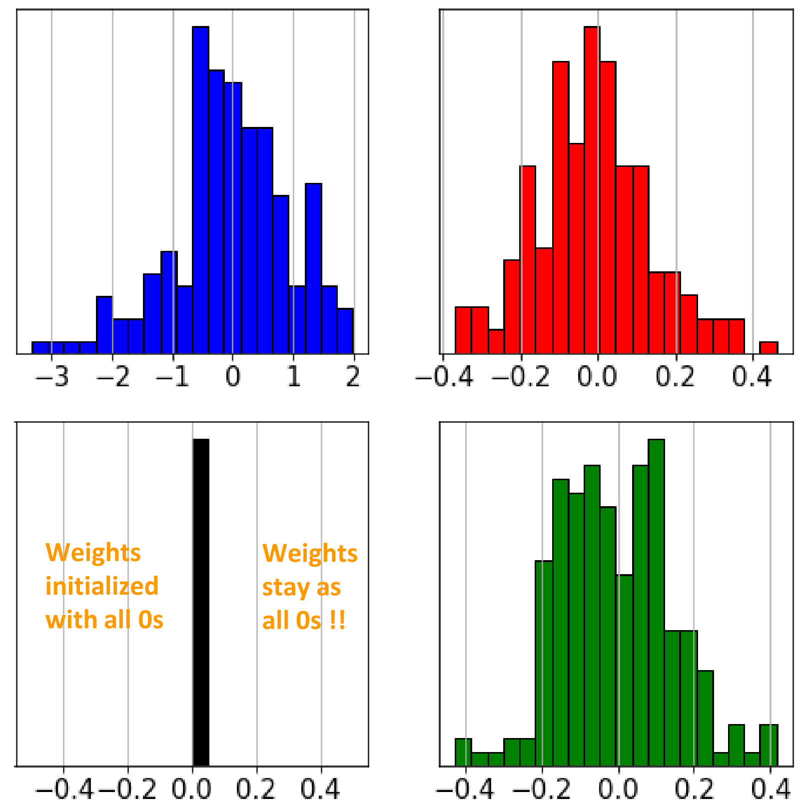
Kaiming Uniform Initialization

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

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"

PYTORCH INITIALIZERS

`torch.nn.init.uniform_(tensor, a=0.0, b=1.0)` [SOURCE]

Fills the input Tensor with values drawn from the uniform distribution $\mathcal{U}(a, b)$.

`torch.nn.init.normal_(tensor, mean=0.0, std=1.0)` [SOURCE]

Fills the input Tensor with values drawn from the normal distribution $\mathcal{N}(\text{mean}, \text{std}^2)$.

`torch.nn.init.constant_(tensor, val)` [SOURCE]

Fills the input Tensor with the value val .

`torch.nn.init.ones_(tensor)` [SOURCE]

Fills the input Tensor with the scalar value *1*.

`torch.nn.init.zeros_(tensor)` [SOURCE]

Fills the input Tensor with the scalar value *0*.

`torch.nn.init.eye_(tensor)` [SOURCE]

Fills the 2-dimensional input *Tensor* with the identity matrix. Preserves the identity of the inputs in *Linear* layers, where as many inputs are preserved as possible.

`torch.nn.init.dirac_(tensor, groups=1)` [SOURCE]

Fills the {3, 4, 5}-dimensional input *Tensor* with the Dirac delta function. Preserves the identity of the inputs in *Convolutional* layers, where as many input channels are preserved as possible. In case of groups>1, each group of channels preserves identity

`torch.nn.init.xavier_uniform_(tensor, gain=1.0)` [SOURCE]

Fills the input *Tensor* with values according to the method described in *Understanding the difficulty of training deep feedforward neural networks* - Glorot, X. & Bengio, Y. (2010), using a uniform distribution. The resulting tensor will have values sampled from $\mathcal{U}(-a, a)$ where

https://pytorch.org/docs/stable/nn.init.html

https://pytorch.org/docs/master/generated/torch.nn.Module.html

`apply(fn: Callable[Module, None]) → T` [SOURCE]

Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch.nn.init).

Example:

```
>>> @torch.no_grad()
>>> def init_weights(m):
>>>     print(m)
>>>     if type(m) == nn.Linear:
>>>         m.weight.fill_(1.0)
>>>         print(m.weight)
>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```