



MIDDLE EAST TECHNICAL UNIVERSITY

DI504

Foundations of Deep Learning

Deep Training

Welcome again!

This is DI504, Foundations of Deep Learning,

- At this point we know how to construct a deep learning model, and practically train it.
- This we talk more about training, and learning.

Backpropagation

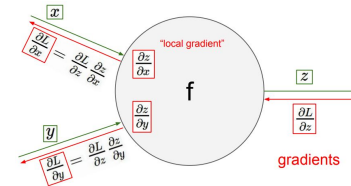
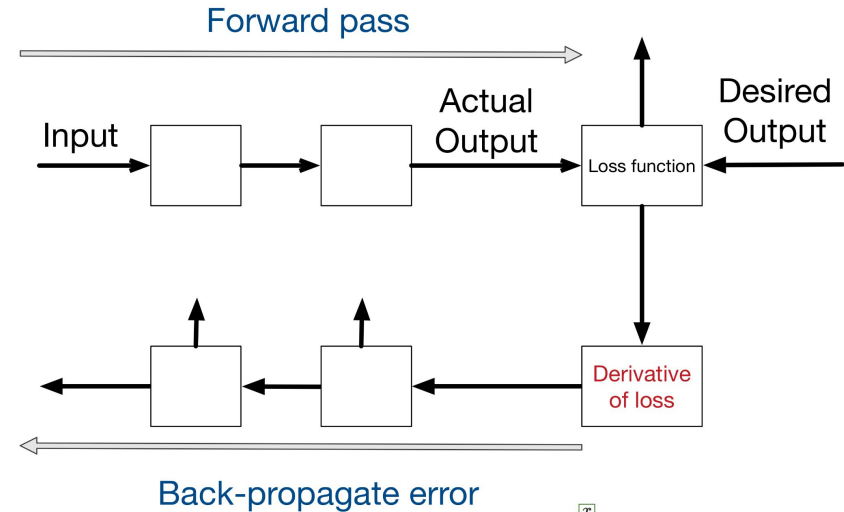
- Back propagation is just an efficient way of computing gradients in computational graphs.
- Update mechanism is the “solver”.

Algorithm 1 Pseudocode for BP algorithm

```

1: procedure BACKPROPAGATION( $\mathcal{D}, \eta$ )
2:   Input:  $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^n$ , learning rate  $\eta$ 
3:   Randomly initialize all weights and threshold
4:
5:   repeat
6:     for all  $(x^{(i)}, y^{(i)}) \in \mathcal{D}$  do
7:       Compute  $y_j^{(i)}$  according current parameter
8:       compute  $\delta_{\beta_i}$ 
9:       Compute  $\delta_{\alpha_j}$ 
10:      update  $w_{ji}, v_{kj}$ 
11:    end for
12:  until achieve stopping condition
13: end procedure
  
```

www.alibabacloud.com



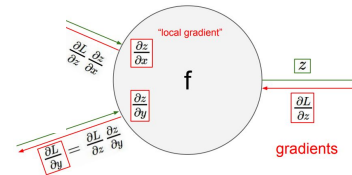
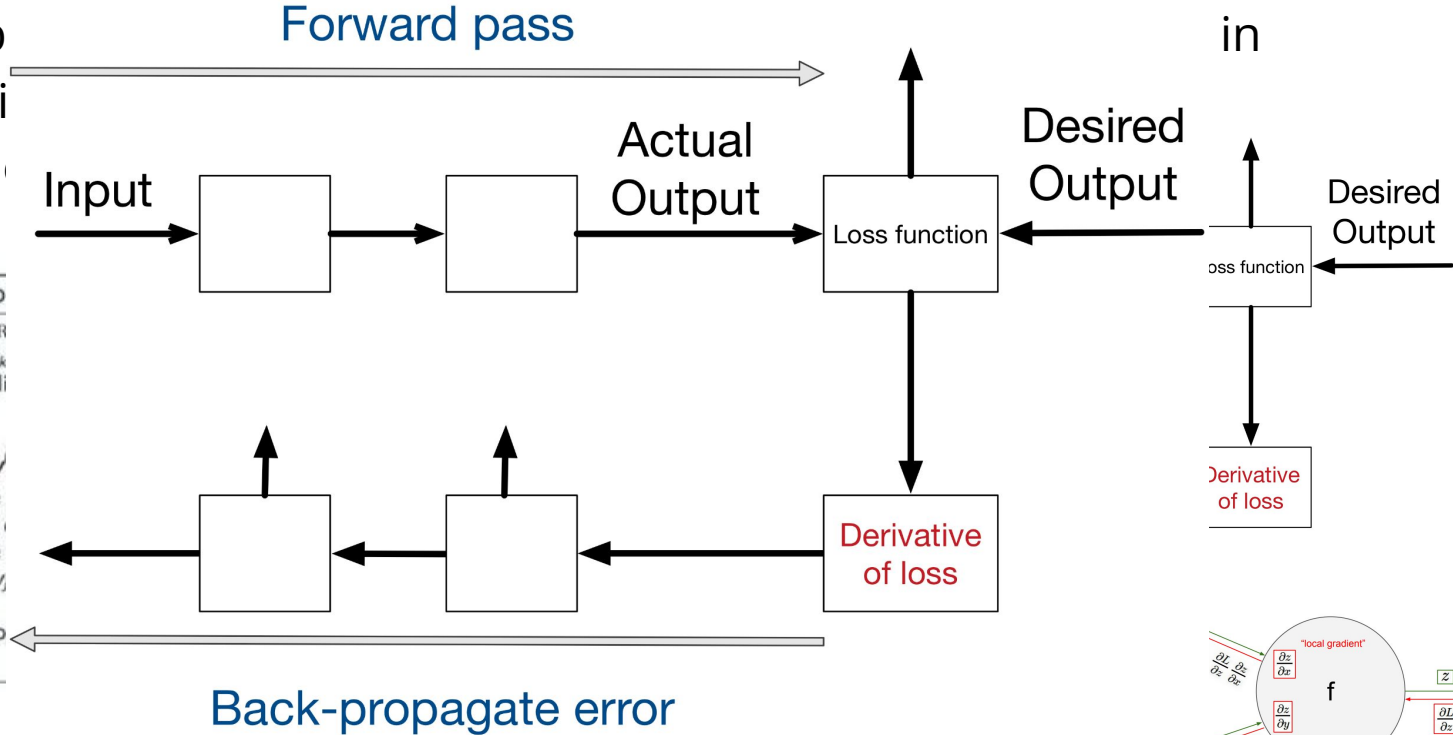
Backpropagation

- Back prop computation
- Update model

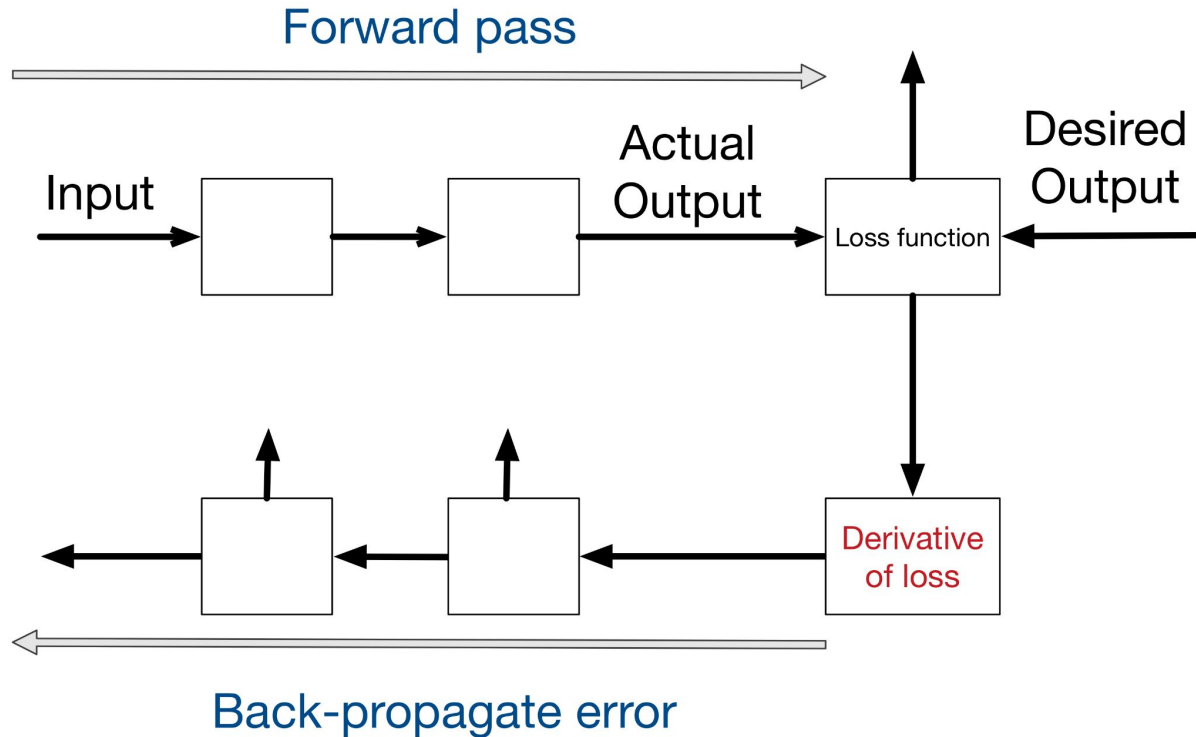
Algorithm 1 Pseudo

```

1: procedure BACKPROP
2:   Input:  $\mathcal{D} = \{(x_k, y_k)\}$ 
3:   Randomly initialize weights
4:
5:   repeat
6:     for all  $(x^{(j)}, y^{(j)}) \in \mathcal{D}$ 
7:       Compute forward pass
8:       compute loss
9:       Compute backward pass
10:      update weights
11:    end for
12:  until achieve stopping criteria
13: end procedure
  
```



Backpropagation





Backpropagation in PyTorch

Autograd in PyTorch is a reverse automatic differentiation system. Conceptually, autograd:

- records your model's computational graph by recording all of the operations that created the data as you execute operations,
- giving you a directed acyclic graph whose leaves are the input tensors and roots are the output tensors.

The screenshot shows a web browser window with the URL `pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html`. The page title is "A GENTLE INTRODUCTION TO TORCH.AUTOGRAD". The text explains that `torch.autograd` is PyTorch's automatic differentiation engine. It describes the background of neural networks and the two steps of training: forward propagation and backward propagation. It also shows a code snippet for using Autograd in PyTorch.

A GENTLE INTRODUCTION TO `TORCH.AUTOGRAD`

`torch.autograd` is PyTorch's automatic differentiation engine that powers neural network training. In this section, you will get a conceptual understanding of how autograd helps a neural network train.

Background

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by *parameters* (consisting of weights and biases), which in PyTorch are stored in tensors.

Training a NN happens in two steps:

Forward Propagation: In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.

Backward Propagation: In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (*gradients*), and optimizing the parameters using gradient descent. For a more detailed walkthrough of backprop, check out this [video from 3Blue1Brown](#).

Usage in PyTorch

Let's take a look at a single training step. For this example, we load a pretrained resnet18 model from `torchvision`. We create a random data tensor to represent a single image with 3 channels, and height & width of 64, and its corresponding `label` initialized to some random values.

```
import torch, torchvision
model = torchvision.models.resnet18(pretrained=True)
data = torch.rand(1, 3, 64, 64)
labels = torch.rand(1, 1000)
```

Backpropagation in PyTorch

PyTorch creates something called a Dynamic Computation Graph, which means that the graph is generated on the fly.

A graph is created on the fly

```
from torch.autograd import Variable  
  
x = Variable(torch.randn(1, 10))  
prev_h = Variable(torch.randn(1, 20))  
W_h = Variable(torch.randn(20, 20))  
W_x = Variable(torch.randn(20, 10))
```



Backpropagation in PyTorch

PyTorch creates something called a Dynamic Computation Graph, which means that the graph is generated on the fly.

Actually all DL frameworks maintain a computational graph that defines the order of computations that are required to be performed.

```
import torch.nn as nn
import torch.nn.functional as F

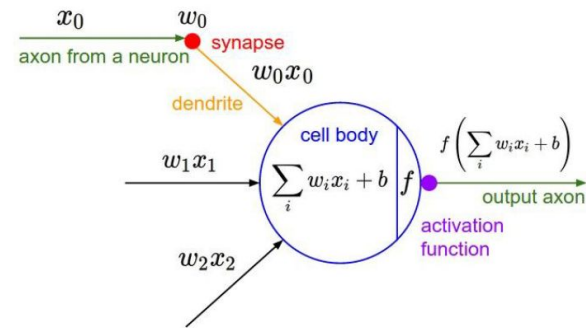
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net()
```

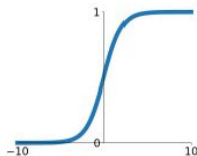

Activation Functions

Why ReLU in AlexNet



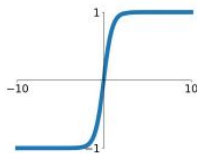
Sigmoid

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



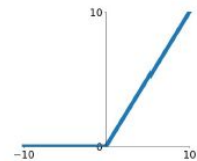
tanh

$$\tanh(x)$$



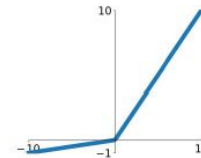
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

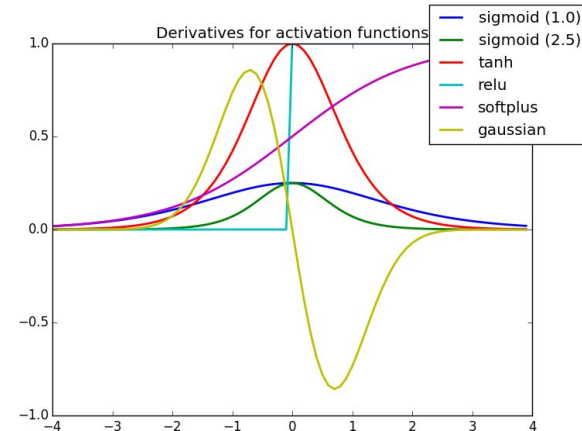
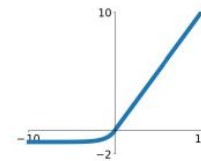


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

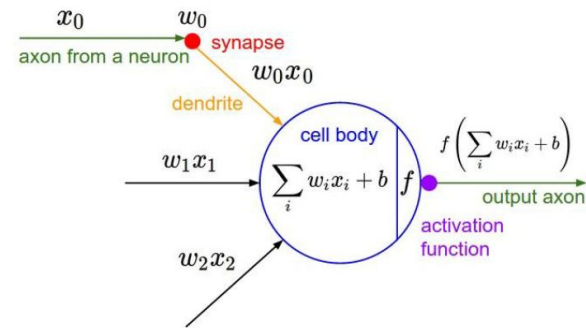
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



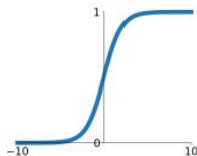
Activation Functions

Why ReLU in AlexNet



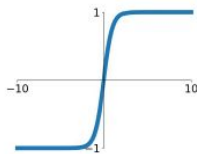
Sigmoid

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



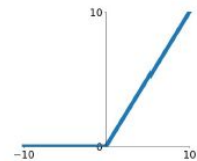
tanh

$$\tanh(x)$$



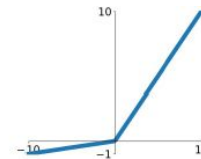
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

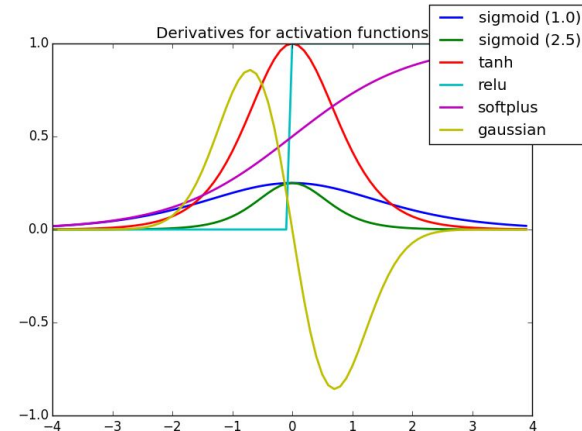
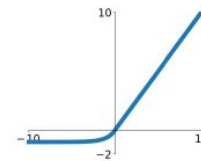


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

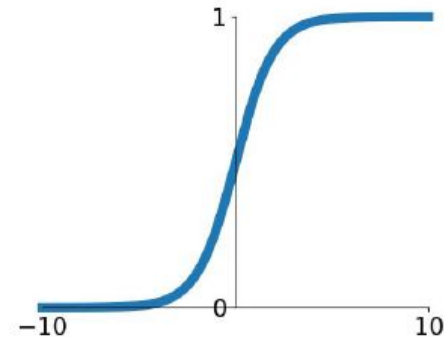


Activation Functions

For example, what about Sigmoid?

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

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



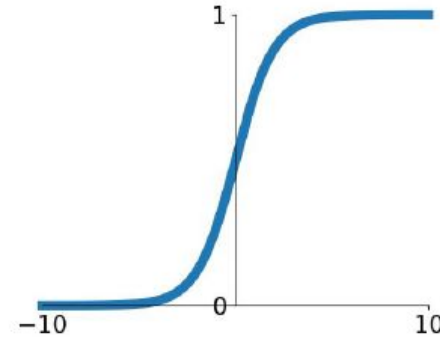
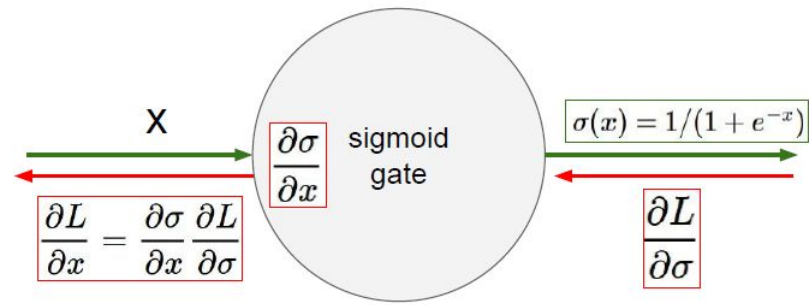
Sigmoid

Activation Functions

For example, what about Sigmoid?

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

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



Sigmoid

3 problems:

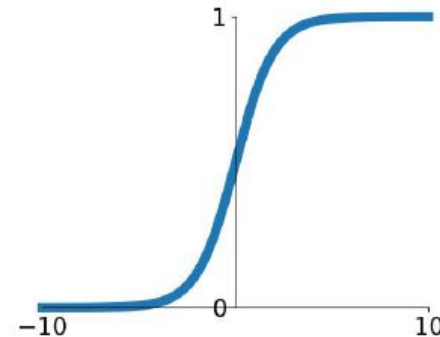
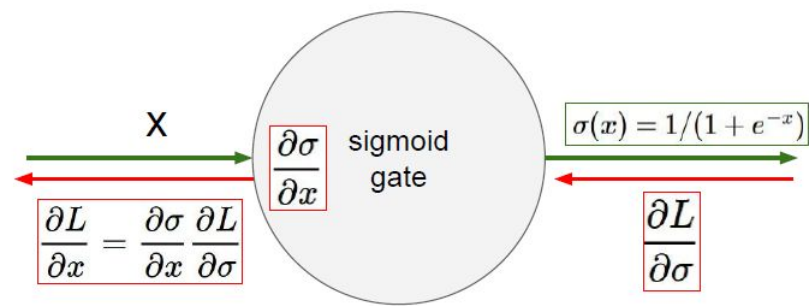
1. Saturated neurons “kill” the gradients

Activation Functions

For example, what about Sigmoid?

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

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



Sigmoid

3 problems:

2. Sigmoid outputs are not zero-centered

Activation Functions

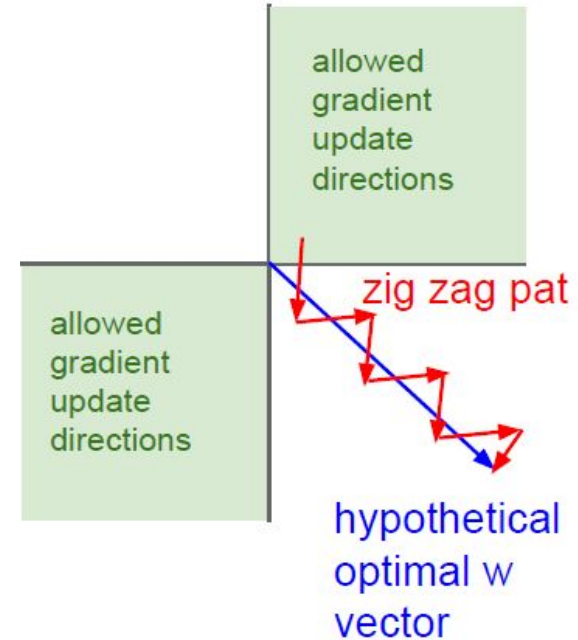
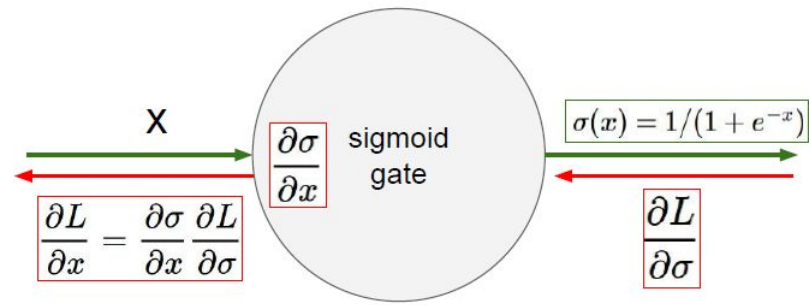
For example, what about Sigmoid?

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

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

3 problems:

1. Sigmoid outputs are not zero-centered
2. Sigmoid outputs are not zero-centered



Activation Functions

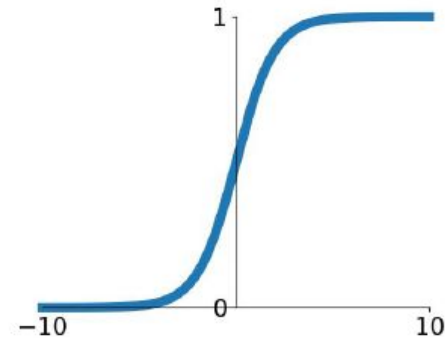
For example, what about Sigmoid?

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

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

3 problems:

3. $\exp()$ is a bit compute expensive

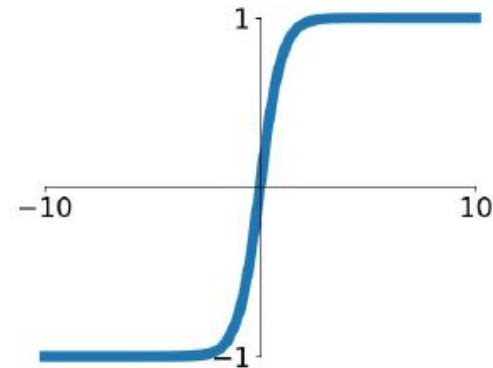


Sigmoid

Activation Functions

For example, what about tanh?

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



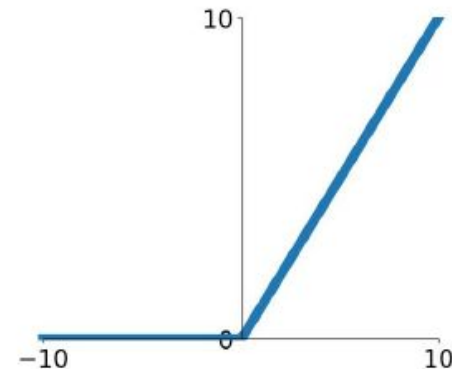
$\tanh(x)$

Activation Functions

For example, what about ReLU?

- ✓ Does not saturate in + region.
- ✓ Very computationally efficient

- Not zero centered though!
- Zero grad for negative input
 - Which is actually a problem in some cases

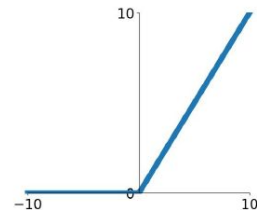


ReLU
(Rectified Linear Unit)

Dying ReLU

Because ReLU outputs 0 for every negative value, a ReLU neuron might get stuck (dead) in the negative side and always output 0, and it is unlikely for it to recover.

If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold.

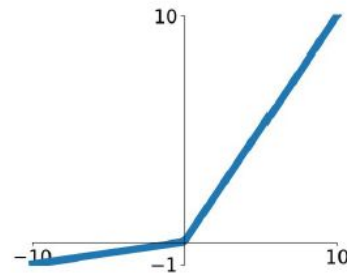


ReLU
(Rectified Linear Unit)

Leaky ReLU

For example, what about Leaky ReLU?

- ✓ Does not saturate in + region.
- ✓ Very computationally efficient
- ✓ Gradients do not die!



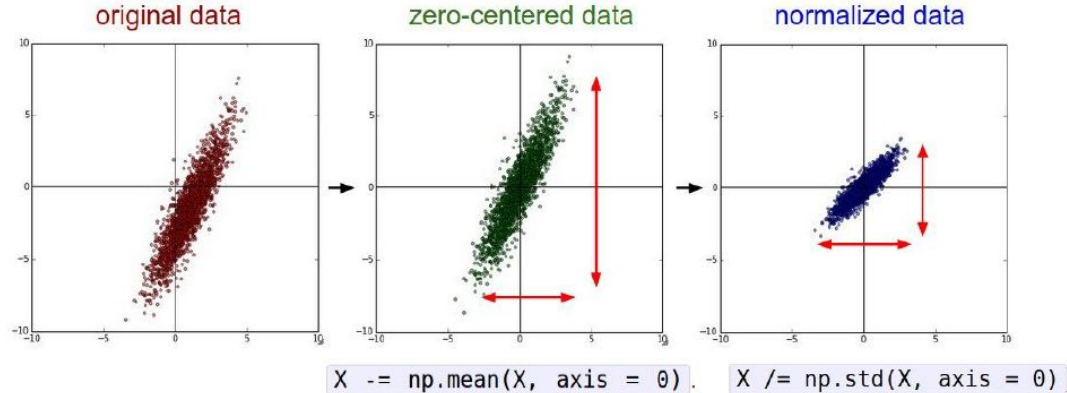
Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Preprocessing

Why normalize?

- The same reason why, sigmoid's not-zero-centered output was a problem.
- Also dimensional differences (in some signals) could create problems.

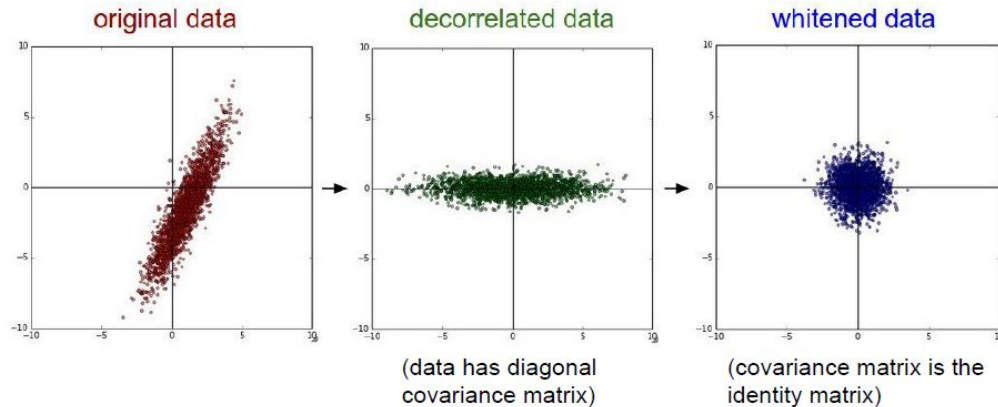


Preprocessing

Why normalize?

- What is more, correlation within the dimension can make optimization difficult.

In practice, you may also see **PCA** and **Whitening** of the data



Weight Initialization

Weight initialization is an important design choice when developing deep learning neural network models.

Historically, weight initialization involved using small random numbers, although over the last decade, more specific heuristics have been developed that use information, such as the type of activation function that is being used and the number of inputs to the node.

These more tailored heuristics can result in more effective training of neural network models using the stochastic gradient descent optimization algorithm.

Weight Initialization

“... training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.”

- page 301, [Deep Learning](#), 2016

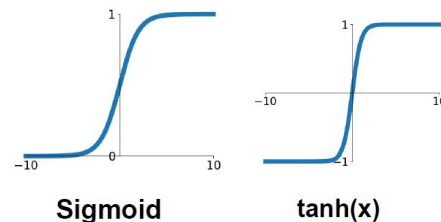
Weight Initialization

Each time, a neural network is initialized with a different set of weights, resulting in a different starting point for the optimization process, and potentially resulting in a different final set of weights with different performance characteristics.

We cannot initialize all weights to zeros as the optimization algorithm results in some asymmetry in the error gradient to begin searching effectively.

For ReLu it is singular value.

Xavier Initialization



The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called “Glorot” or “Xavier” initialization.

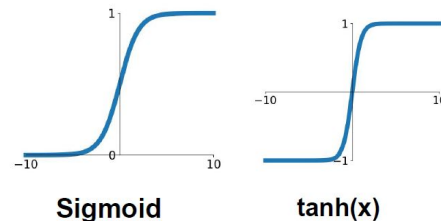
It is named for Xavier Glorot and was described in the 2010 paper by Xavier and Yoshua Bengio titled “[Understanding The Difficulty Of Training Deep Feedforward Neural Networks.](#)”

Glorot and Bengio proposed to adopt a properly scaled uniform distribution for initialization. This is called “Xavier” initialization.

*a random number with a uniform probability distribution,
n is the number of inputs*

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

Xavier Initialization



The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called “Glorot” or “Xavier” initialization.

It is named for Xavier Glorot and was described in the 2010 paper by Xavier and Yoshua Bengio titled “[Understanding The Difficulty Of Training Deep Feedforward Neural Networks.](#)”

Glorot and Bengio proposed to adopt a properly scaled uniform distribution for initialization. This is called “Xavier” initialization.

There is also a normalized version:

$$W \sim U \left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}} \right]$$

Xavier Initialization

Xavier init. Works well with tanh and sigmoid.

But it was found to have problems when used to initialize networks that use the rectified linear (ReLU) activation function.

As such, a modified version of the approach was developed specifically for nodes and layers that use ReLU activation, popular in the hidden layers of most multilayer Perceptron and convolutional neural network models.

So what about ReLU?


He Initialization

The current standard approach for initialization of the weights of neural network layers and nodes that use the rectified linear (ReLU) activation function is called “he” initialization.

It is named for Kaiming He, currently a research scientist at Facebook, and was described in the 2015 paper by Kaiming He, et al. titled “[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.](#)”

He Initialization

The He initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of $\sqrt{2/n}$, where n is the number of inputs to the node.

$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$


this is variance

This formula is valid only when we use ReLU in each layer.

Babysitting the Training!

I call the first stage the “the Training”, since we only use the training data.

- Preprocess (normalize) the input data
- Initialize each layer weights according to their activation functions.
- Check the loss after the first forward pass for a reasonable value
(*i.e. not NaN*)
- Observe the training loss. Log it. Visualise it. Is it decreasing?
(*Your first goal is to overfit the training data.*)
- If training loss do not fall, check the activations? Dead neurons maybe?

Babysitting the Training!

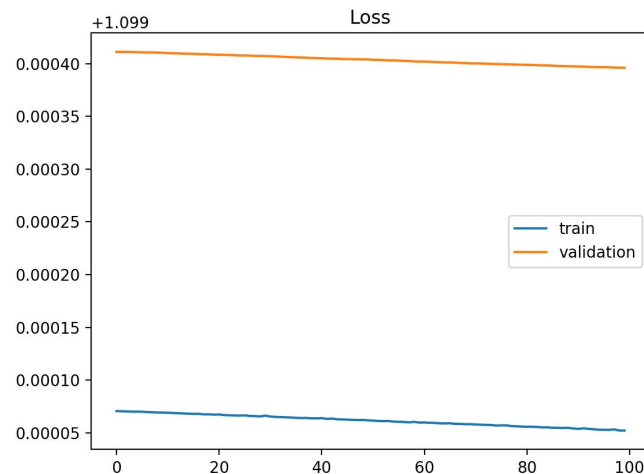
I call the second stage the “the Validation”, since we will start checking the validation data at this point

- Optimize your hyper parameters!
- Play with:
 - Network architecture (# filters etc.)
 - Learning rate, decay rate (soon)
 - Regularization (if overfitting too bad)
- And observe training vs validation losses/accuracies.

Training Graphs

Underfitting

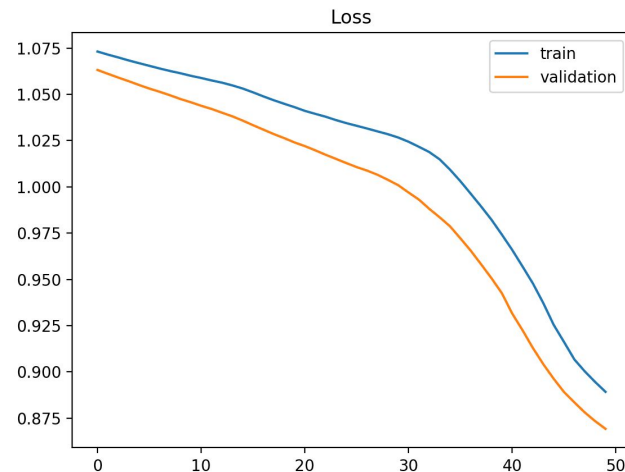
- An underfit model can be identified from the learning curve of the training loss only.
- It may show a flat line or noisy values of relatively high loss, indicating that the model was unable to learn the training dataset at all.
- An example of this is shown in the figure and is common when the model does not have a suitable capacity for the complexity of the dataset.



Training Graphs

Underfitting

- An underfit model may also be identified by a training loss that is decreasing and continues to decrease at the end of the plot.
- This indicates that the model is capable of further learning and possible further improvements and that the training process was halted prematurely.



Training Graphs

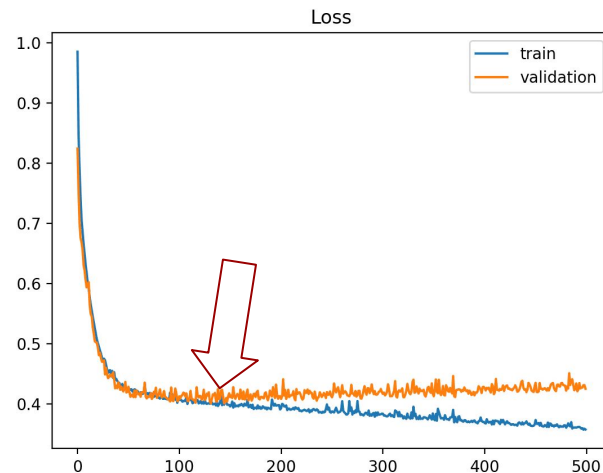
Overfitting

- Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset.
- The problem with overfitting, is that the more specialized the model becomes to training data, the less well it is able to generalize to new data, resulting in an increase in generalization error. This increase in generalization error can be measured by the performance of the model on the validation dataset

Training Graphs

Overfitting

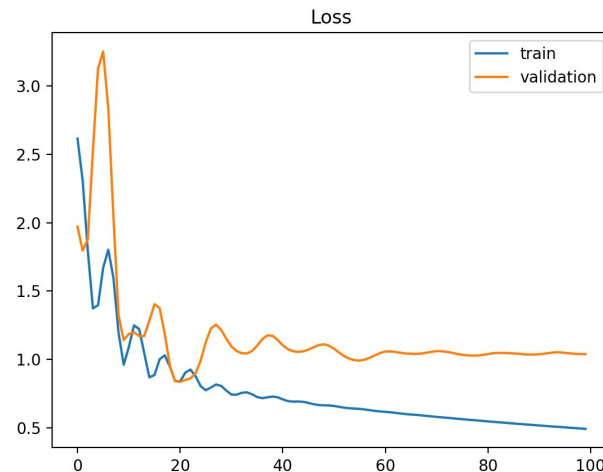
- A plot of learning curves shows overfitting if:
 - The plot of training loss continues to decrease with experience.
 - The plot of validation loss decreases to a point and begins increasing again.
- The *inflection point* in validation loss may be the point at which training could be halted as experience after that point shows the dynamics of overfitting.



Training Graphs

Unrepresentative Train Dataset

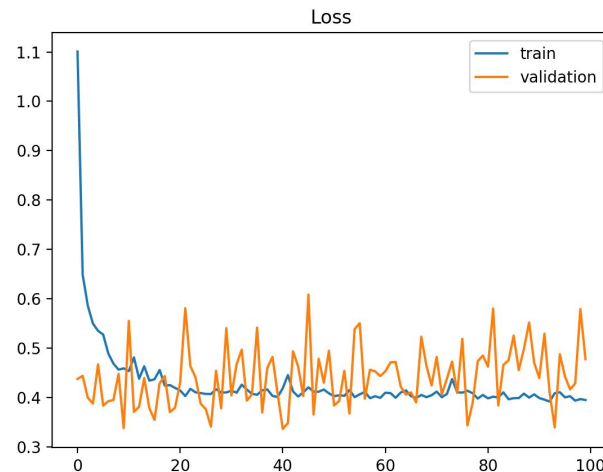
- An unrepresentative training dataset means that the training dataset does not provide sufficient information to learn the problem, relative to the validation dataset used to evaluate it.
- This may occur if the training dataset has too few examples as compared to the validation Dataset.
- This is similar to overfitting.



Training Graphs

Unrepresentative Validation Dataset

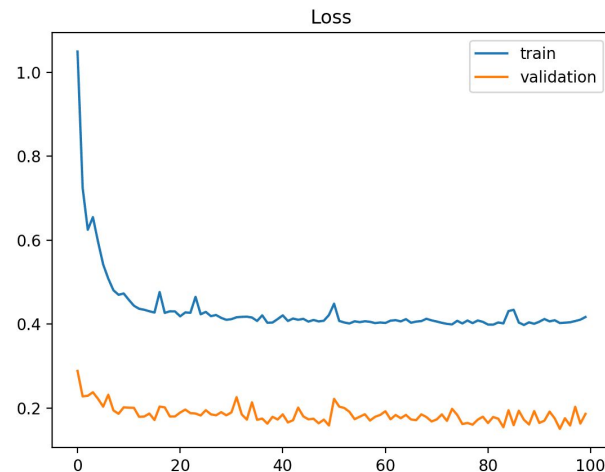
- An unrepresentative validation dataset means that the validation dataset does not provide sufficient information to evaluate the ability of the model to generalize.
- This may occur if the validation dataset has too few examples as compared to the training dataset.
- This case can be identified by a learning curve for training loss that looks like a good fit (or other fits) and a learning curve for validation loss that shows noisy movements around the training loss.



Training Graphs

Unrepresentative Validation Dataset

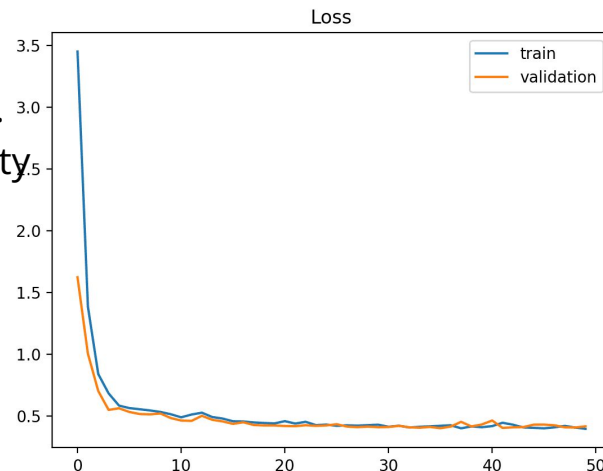
- It may also be identified by a validation loss that is lower than the training loss. In this case, it indicates that the validation dataset may be easier for the model to predict than the training dataset.



Training Graphs

Good Fit

- A good fit is the goal of the learning algorithm and exists between an overfit and underfit model.
- There is no perfect fit.
- A plot of learning curves shows a good fit if:
 - The plot of training loss decreases to a point of stability.
 - The plot of validation loss decreases to a point of stability and has a small gap with the training loss.



Solvers

(Gradient Descent first)

Gradient Descent fundamentals were:

- Score function : $f(x_i, \theta) = y_i$
- Ground Truth : g_i
- Loss function : $L(x_i, g_i) = g[f(x_i, \theta), g_i]$
- Gradient Descent :
$$\theta_{\text{new}} = \theta_{\text{old}} - \lambda \frac{\delta L(x_i, g_i, \theta_{\text{old}})}{\delta \theta_{\text{old}}}$$

Gradient Descent

Gradient Descent is a good idea, but it is immature, hence, a bit problematic.

The main problem with gradient descent is that the weight update at a moment (t) is governed by the learning rate and gradient at that moment only. It doesn't take into account the past steps taken while traversing the cost space

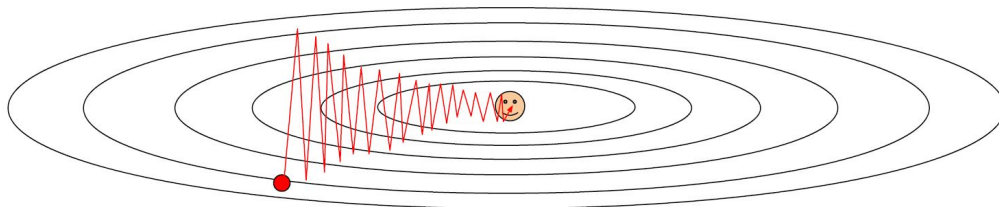
$$\theta_{\text{new}} = \theta_{\text{old}} - \lambda \frac{\delta L(x_i, g_i, \theta_{\text{old}})}{\delta \theta_{\text{old}}}$$

Gradient Descent

GD has additional problems as well.

Mathematically speaking, if the loss function has high condition number:
(ratio of largest to smallest singular value of the Hessian matrix is large)

Or in other words, if the valley is full of small pits, then:



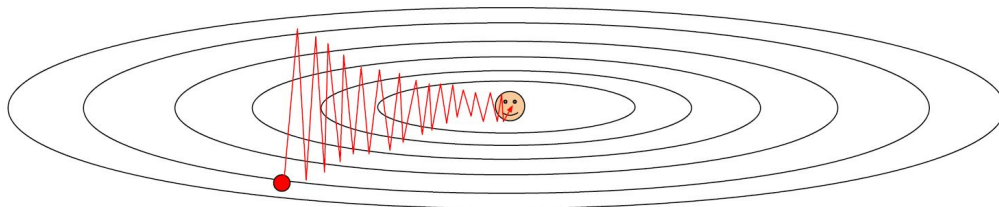
$$\theta_{\text{new}} = \theta_{\text{old}} - \lambda \frac{\delta L(x_i, g_i, \theta_{\text{old}})}{\delta \theta_{\text{old}}}$$

Gradient Descent

GD has additional problems as well.

Mathematically speaking, if the loss function has high condition number:
(ratio of largest to smallest singular value of the Hessian matrix is large)

Or in other words, if the valley is full of small pits, then:



$$\theta_{\text{new}} = \theta_{\text{old}} - \lambda \frac{\delta L(x_i, g_i, \theta_{\text{old}})}{\delta \theta_{\text{old}}}$$

Stochastic Gradient Descent

And remember that, we apply Stochastic Gradient Descent, which is simply loss being calculated collectively for a minibatch of inputs.

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

And this fact may also create noisy loss functions too (i.e. bumpy valley).

Stochastic Gradient Descent + Momentum

So the first thing we may add is momentum.

It is simply, building up “velocity” as a running mean of gradients.
(i.e. Blind guy remembering what he was doing for a while)

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
    dx = compute_gradient(x)
    x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x += learning_rate * vx
```

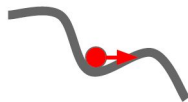
Stochastic Gradient Descent + Momentum

So the first thing we may add is momentum.

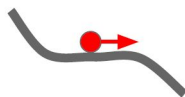
It is simply, building up “velocity” as a running mean of gradients.

(i.e. Blind guy remembering what he was doing for a while)

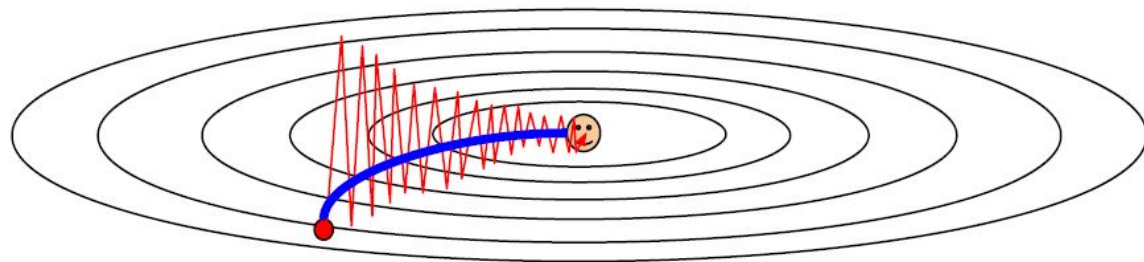
Local Minima



Saddle points



SGD+Momentum





Solvers

(also called optimizers in PyTorch)

There are many different gradient descent-based optimizers, with minor modifications.

- [Adadelta](#), [Adagrad](#), [Adam](#), [AdamW](#), [SparseAdam](#), [Adamax](#), [ASGD](#), [LBFGS](#), [NAdam](#), [RAdam](#), [RMSprop](#), [Rprop](#), [SGD](#)

Each optimizer is known to have pros and cons in various conditions.



Solvers - How to pick?

Adam is a good choice in most cases.

Adam was presented by [Diederik Kingma](#) from OpenAI and [Jimmy Ba](#) from the University of Toronto in their 2015 ICLR paper (poster) titled "[Adam: A Method for Stochastic Optimization](#)".

Benefits of using Adam on non-convex optimization problems, as follows:

- Straightforward to implement.
- Computationally efficient.
- Little memory requirements.
- Invariant to diagonal rescale of the gradients.
- Well suited for problems that are large in terms of data and/or parameters.
- Appropriate for non-stationary objectives.
- Appropriate for problems with very noisy/or sparse gradients.
- Hyper-parameters have intuitive interpretation and typically require little tuning.

Ready to train?

I hope so. You have:

- A dataset with ground truth (labels, annotations, etc.)
- A model architecture
- A good optimizer/solver

Good to go then? But is your data sufficiently large in scale?

No? Ok. Here comes data augmentation.

Data Augmentation

How do I get more data, if I don't have "more data"?

You don't need to hunt for novel new images that can be added to your dataset. Because, neural networks aren't that smart to begin with.

For example, a poorly trained neural network would think that these three tennis balls shown below, are distinct, unique images.



Data Augmentation

Data augmentation in data analysis is a group of techniques used to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data.

It acts as a regularizer and helps reduce overfitting when training a machine learning model.

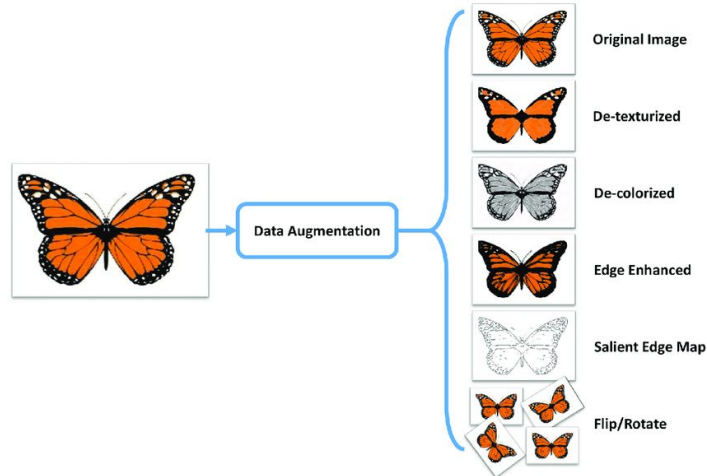
It is closely related to *oversampling* in data analysis.

Data Augmentation

The augmentation technique to be applied depends on:

- Input modality (image, sound, text, etc.)
- The problem (classification, segmentation, machine translation, etc.)

Vision:



Data Augmentation

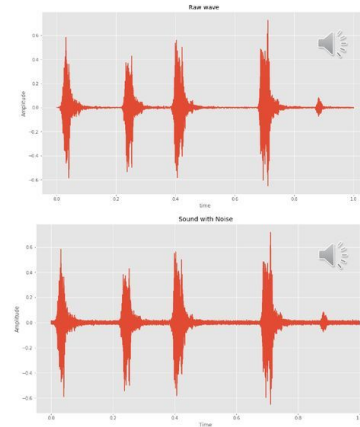
The augmentation technique to be applied depends on:

- Input modality (image, sound, text, etc.)
- The problem (classification, segmentation, machine translation, etc.)

Audio:

■ Types of Audio data augmentation:

- (1) Time stretching
- (2) Pitch Shifting
- (3) Dynamic range compression
- (4) Background Noise



Data Augmentation

The augmentation technique to be applied depends on:

- Input modality (image, sound, text, etc.)
- The problem (classification, segmentation, machine translation, etc.)

NLP:



Transfer Learning

Still not enough data!

Transfer learning (TL) is a concept in ML that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem.

For example, knowledge gained while learning to recognize cars could apply when trying to recognize trucks.

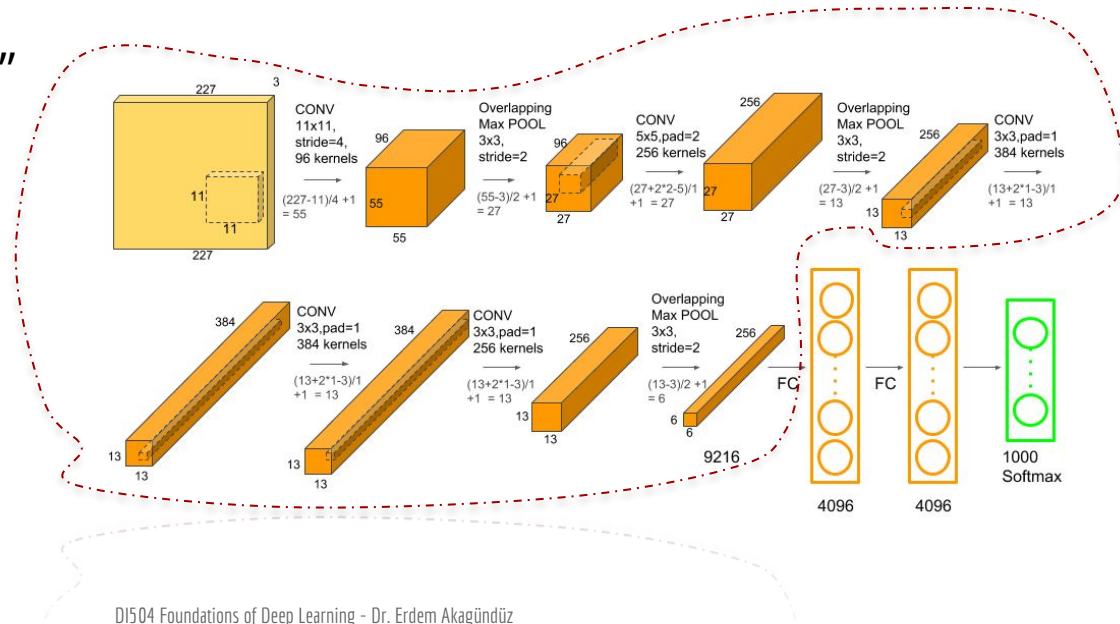
In deep learning it is usually applied to deep learning for **feature encoding**.

Transfer Learning

Remember AlexNet

Conv layers were used to encode visual features.

We can use this “encoder” in another architecture by “transferring” these Layers, with their pretrained weights.



Transfer Learning

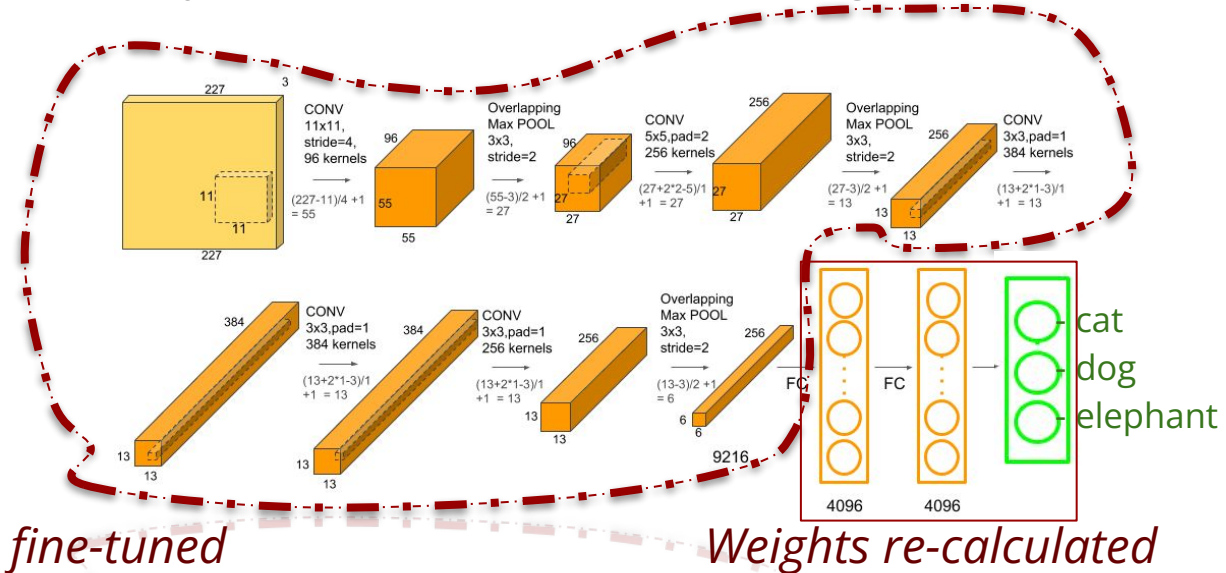
Remember AlexNet

Maybe we can change the final layer (1000 nodes for 1000 categories) to a 3 node layer to classify

- cat
- dog
- elephant, images

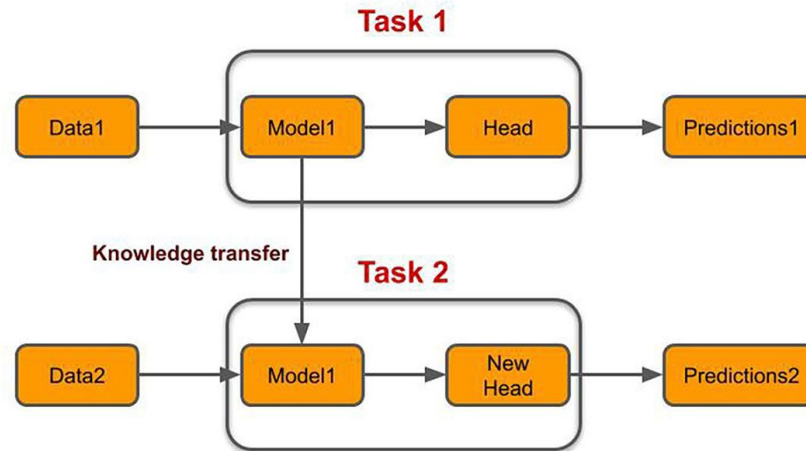
(if that is what we are up to).

**But I don't have enough cat,
dog, elephant images !**



Transfer Learning

- Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.



Transfer Learning

- DL models (although we have ways to analyse/visualise them) are not transparent.
- So where to transfer, which conv layers to get, what architecture to transfer from?
- How to set the learning rate for fine-tuning or learning from scratch? Or maybe freeze some layers?



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

What will we do next week?

- The Midterm!
- Following weeks
 - Introduction to Sequence Models, RNNs
 - LSTMs, and many applications of deep learning.

Additional Reading & References

- <https://towardsdatascience.com/pytorch-autograd-understanding-the-heart-of-pytorchs-magic-2686cd94ec95>
- https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html
- <https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/>
- <https://blog.paperspace.com/pytorch-101-understanding-graphs-and-automatic-differentiation/>
- <https://medium.com/intuitionmachine/pytorch-dynamic-computational-graphs-and-modular-deep-learning-7e7f89f18d1>
- <https://www.v7labs.com/blog/neural-networks-activation-functions>
- <https://medium.com/@shubham.deshmukh705/dying-relu-problem-879cec7a687f>
- <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/>
- <https://machinelearningmastery.com/different-results-each-time-in-machine-learning/>
- <https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/>
- <http://proceedings.mlr.press/v9/glorot10a.html>
- <https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79>
- <https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/>
- <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- <https://machinelearningmastery.com/transfer-learning-for-deep-learning/>