Fundamentals of Machine Learning:

Introduction to Deep Learning

Prof. Andrew D. Bagdanov (`andrew.bagdanov AT unifi.it`)

UNIVERSITÀ DEGLI STUDI FIRENZE

DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Outline

Introduction

Connectionism: The Old School

Connectionism: The New School

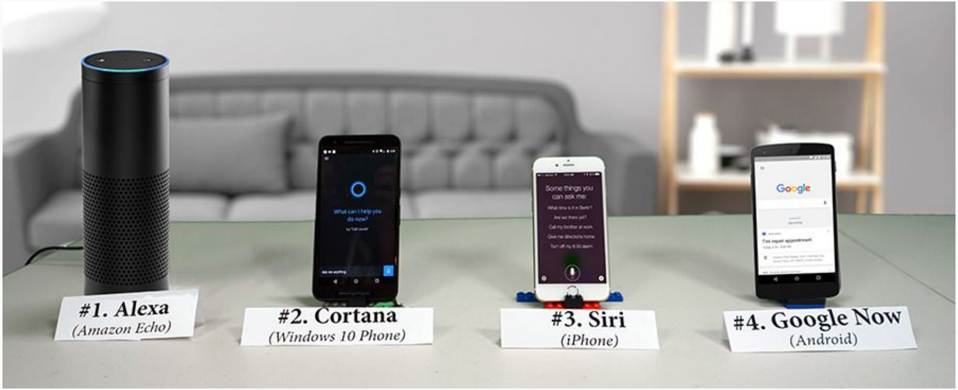
Building and Training Deep Networks

Discussion

Introduction

Digital assistants

- Deep learning is profoundly changing our lives.



Natural language processing

- Deep Recurrent Neural Networks (RNNs) are powering the latest generation of natural language translation technologies.

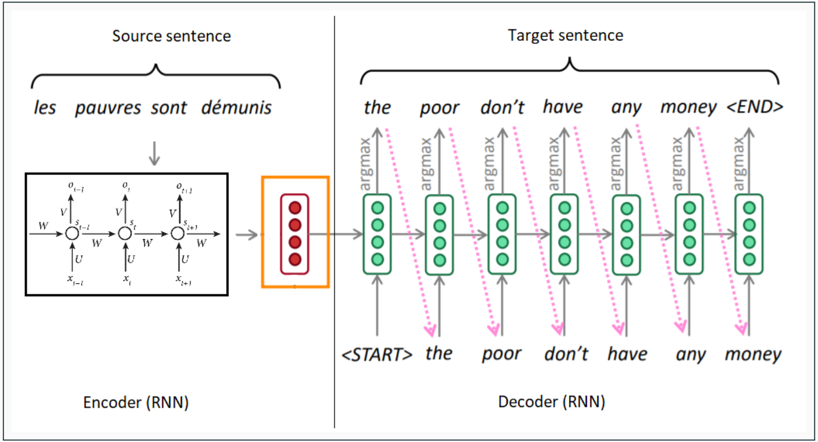
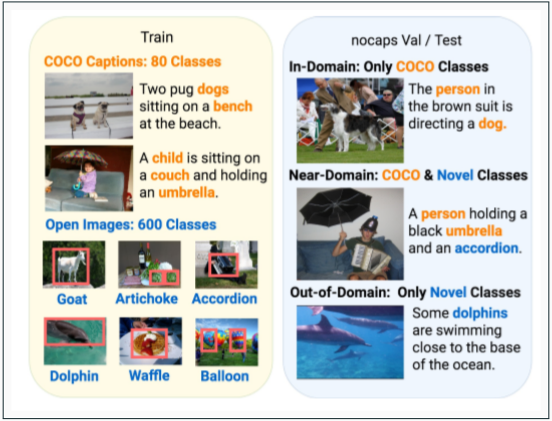


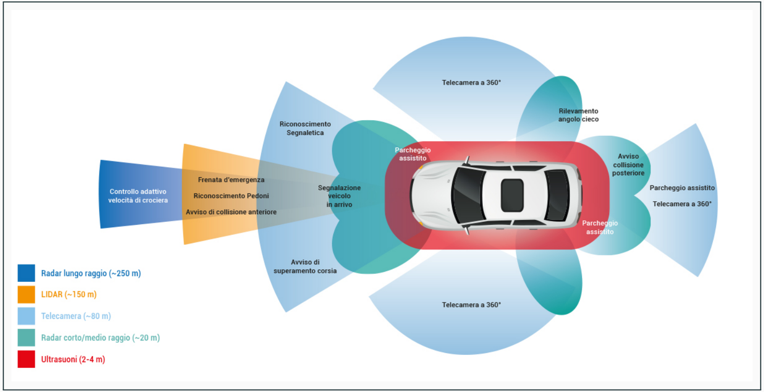
Image captioning

- Convolutional Neural Networks (CNNs) are able to extract high-level semantics from images.



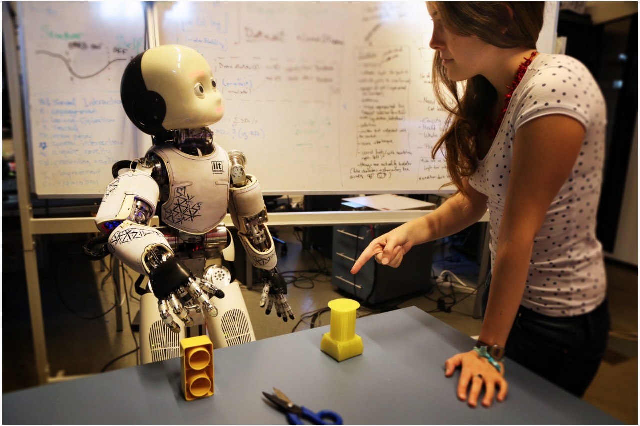
Self-driving cars

- CNNs are able to integrate multi-modal inputs and are driving the latest advances in Automatic Driving Assistance (ADAS) systems.



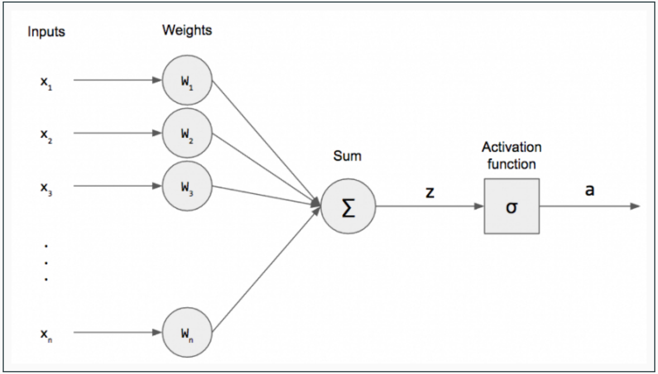
Reinforcement learning

- Deep Reinforcement Learning is being used to train robots who can learn from experience and interactions with their environment.



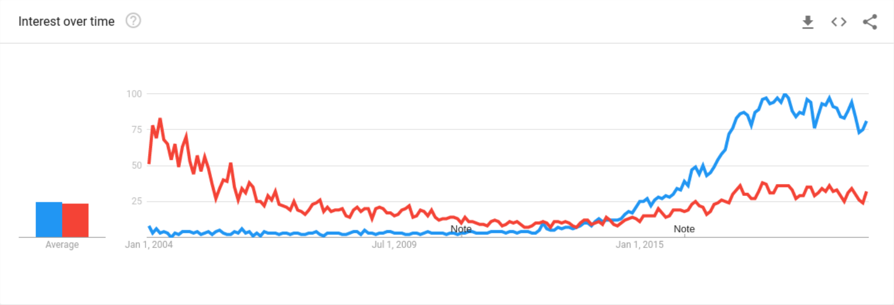
All thanks to...

- The humble Neural Network.
- Artificial Neural Networks (ANNs) are extremely simple, yet also extremely powerful models.
- They are, in fact, universal function approximators.



Neural Networks are not new

- As we will see, neural networks have a storied history.
- Deep Learning, however, is their modern incarnation.



Overview

- Today we will see what puts the deep into Deep Learning.
- We will start with an overview of some historical milestones in the development of artificial neural networks.
- Then we will look at how modern deep neural networks are actually built:
- We will see how the basic Multilayer Perceptron (MLP) model provides a modular architecture for machine learning problems.
- And we will see how modern tools (e.g. PyTorch) makes it easy to apply Deep Models to new problems.

Lecture objectives

After this lecture you will:

- Understand what connectionist models are and how they allow us to compute nonlinear functions of inputs.
- Understand how the perceptron works and how its parameters are estimated via error correction.
- Understand how the multi-layer perceptron generalized the perceptron and how it uses input, hidden, and output layers to represent feedforward computations of inputs to outputs.

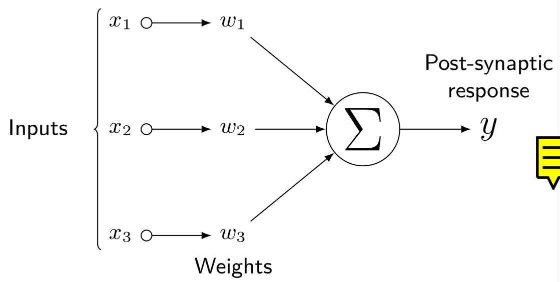
Connectionism: The Old School

What is a Deep Neural Network?

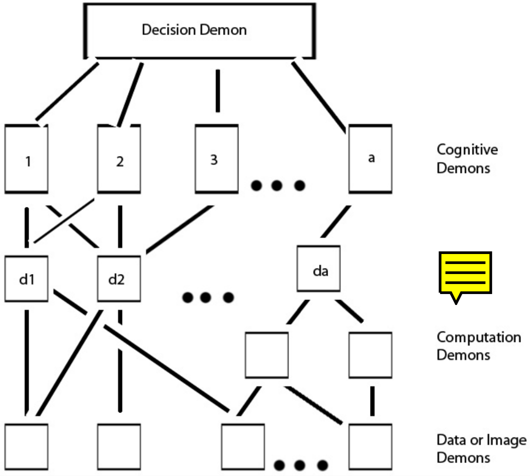
- Neural Networks are connectionist models.
- Connectionism has deep roots reaching back to Classical Greece.
- To understand this rich inheritance it is useful to go back in time and trace the roots of modern Deep Models.
- Connectionism arose from the neuroscience and psychological research communities of the 1940s and 1950s.
- These were the nascent beginning of what would become Cognitive Science.
- Though founded on solid experimental practice, what was lacking was any sort of computation basis for learning.

- One of the first concrete learning rules for connectionist models (both artificial and biological).
- Hebb's Rule: if cell A consistently contributes to the activity of cell B, then the synapse from A to B should be strengthened.
- More quaintly: *neurons that fire together, wire together; neurons that fire out of sync, fail to link.*

$$
\begin{aligned}
w_{ij} &= x_i x_j \\
w_{ij} &= \frac{1}{N} \sum_p x_i^p x_j^p \\
\Delta w_i &= \eta x_i y \\
&= \eta x_i \sum_j w_j x_j
\end{aligned}
$$



Connectionism: The Pandemonium Model



- In 1958 Selfridge proposed a multi-layer, parallel model of machine learning.
- The model consists of four layers, each inhabited by demons.
- Network architecture fixed *a priori*, connections updated using supervised learning.
- Demons yell upwards, higher-level ones listen and respond.
- High-worth demons can replace low-worth ones via combination.

Connectionism: The Perceptron

- The Perceptron is probably the simplest (and most famous) feedforward neural network.
- The perceptron algorithm was invented by Rosenblatt in 1958.
- It was designed to be a machine, and its original purpose was to perform image recognition.

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The perceptron algorithm

Input: $D = \{ (\mathbf{x}_i, y_i) \}_{i=1}^{N}$ (training data)
Output: learned weights $\mathbf{w}$
$\mathbf{w}_0 \leftarrow$ random initialization
$t \leftarrow 1$
while not converged **do**
    for $(\mathbf{x}, y) \in D$ **do**

$$\hat{y} = f(\mathbf{w}^T \mathbf{x})$$

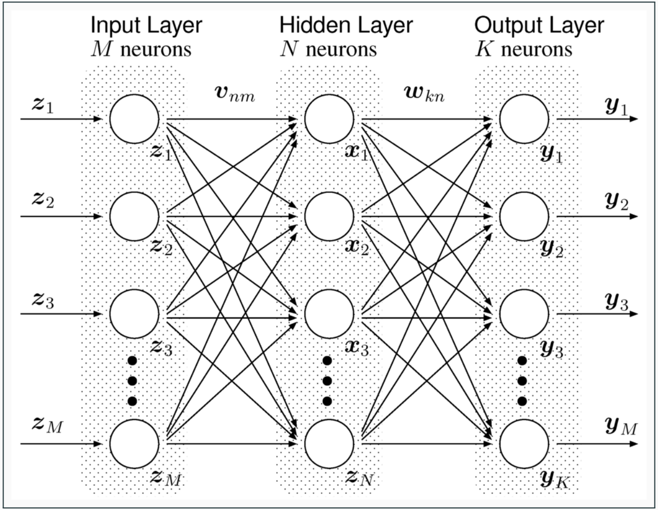
$$\mathbf{w}_t \leftarrow w_{t-1} + \eta(y - \hat{y})\mathbf{x}$$

$$t \leftarrow t + 1$$

Connectionism: The New School

The Multilayer Perceptron

- Let's look at a simple Neural Network architecture known as the Multilayer Perceptron (MLP):



The Multilayer Perceptron (continued)

- The MLP equation (one hidden layer):

$$\hat{\mathbf{y}}(\mathbf{x}) = \sigma(\mathbf{w}_2^T \sigma(\mathbf{w}_1^T \mathbf{x} + b_1) + b_2)$$

- Except for the activation function σ, this is a linear system.
- Common activation functions (elementwise):
  - $\sigma(\mathbf{x}) = \tanh(\mathbf{x})$
  - $\sigma(\mathbf{x}) = (1 + e^{-\mathbf{x}})^{-1}$
  - $\sigma(\mathbf{x}) = \frac{\exp(\mathbf{x})}{\sum_i e^{x_i}}$ (softmax, used for outputs).

Training an MLP

- How do you train a model?
- Decide on a loss function (like the negative log-likelihood):

$$L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) = -\frac{1}{C}\sum_i y_i \log(\hat{y}_i)$$

- And perform gradient descent w.r.t. all model parameters:

$$\begin{aligned}\boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x})) \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n - \varepsilon \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x}_i))\end{aligned}$$

- Where ε is the learning rate.
- The standard algorithm for this is known as backpropagation and it is very clever and efficient.

Training an MLP (continued)

- OK, we perform gradient descent like this:

$$\boldsymbol{\theta}_{n+1} \quad = \quad \boldsymbol{\theta}_n - \varepsilon \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathrm{y}, \hat{\mathrm{y}}(\mathrm{x}_i))$$

Training an MLP (continued)

- OK, we perform gradient descent like this:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \varepsilon \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathrm{y}, \hat{\mathrm{y}}(\mathrm{x}_i))$$

- But... How do I compute that gradient $\nabla_{\boldsymbol{\theta}} L(\mathrm{y}, \hat{\mathrm{y}}(\mathrm{x}_i))$ for non-trivial models?

Training an MLP (continued)

- OK, we perform gradient descent like this:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \varepsilon \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x}_i))$$

- But... How do I compute that gradient $\nabla_{\boldsymbol{\theta}} L(\mathbf{y}, \hat{\mathbf{y}}(\mathbf{x}_i))$ for non-trivial models?
- The standard algorithm for this is known as backpropagation and it is very clever and efficient.
- But... We will defer a detailed discussion of backpropagation for the next lecture.

Stochastic Gradient Descent (SGD)

- Problem: what happens if N (the number of training samples) is very large?
- Well, we end up taking very slow steps – each iteration of gradient descent is an average over the entire dataset.
- Solution: approximate the true gradient with the gradient at a single training example:

Online Stochastic Gradient Descent

- Choose an initial vector of parameters θ and learning rate η.
- Repeat until an approximate minimum is found:
  1. Randomly shuffle training samples in D.
  2. For $(\mathbf{x}, y) \in D$:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\{\mathbf{x}, y\}; \boldsymbol{\theta})$$

Stochastic Gradient Descent (continued)

- Another problem: evaluating the gradient on single examples leads to very noisy steps in parameter space.
- One trick to mitigate this is to use momentum: keep a running average of gradients that is slowly updated.
- Another solution is to use mini-batches: instead of a single sample, average the gradients over a small batch of samples.
- It is common to use a combination of mini-batches and momentum to stabilize training.

SGD Terminology

- Some useful terminology for deep learning optimization:
  - 1 epoch: one complete pass over the data.
  - 1 iteration: a single gradient step.
  - N: number of training samples.
  - B: batch size.

Algorithm	iterations per epoch
Batch gradient descent	1
Stochastic Gradient Descent	N
Mini-batch Gradient Descent	$\frac{N}{B}$

Building and Training Deep Networks

The ingredients

- Network Definition: A neural network model must be defined that precisely describes the transformation from input to output (e.g. a Multi-layer Perceptron with one hidden layer). The model is typically – bu not necessarily – defined in terms of pre-defined, composable modules.
- Dataset and DataLoader: To train a model, we need some data (duh). Data management is critical in Deep Learning, and a DataLoader is responsible for loading, transforming, and batching data for training and inference.
- A Training Loop: Neural Networks are not black boxes, and their training can be delicate and subtle. A training loop applies an iterative optimization algorithm over training batches.

Defining a network: Basics

- A network architecture defines the family of functions we use as models.
- It explicitly defines the parameterized family of functions $\mathcal{H}$ we are searching over.
- An architecture is a Directed Acyclic Graph (DAG) defining the connections between basic computational blocks.
- We call these blocks layers and they are reusable building blocks.

Defining a network: The PyTorch Tensor

- Tensors are a specialized data structure similar to arrays and matrices.
- In PyTorch, tensors are used to:
  - Encode the inputs and outputs of a model.
  - Encode the parameters of a model.
- Tensors vs. NumPy `ndarrays`:
  - Tensors can run on GPUs or other hardware accelerators.
  - Tensors and NumPy arrays can share the same underlying memory, eliminating data copying.
- Tensors are optimized for automatic differentiation.

Defining a network: Tensor Operations

- PyTorch tensors support over 100 tensor operations for:
    - Arithmetic
    - Linear algebra
    - Matrix manipulation (transposing, indexing, slicing)
    - Sampling
    - And much, much more.
- Operations can be run on GPU (typically faster than CPU)
    - Default tensor creation on CPU
    - Move to GPU using .to method (check GPU availability first)

Defining a network: The tedious way

- We can define our network model completely using low-level tensors.
- For linear regression it might look something like this:

```
# Define our model parameters, initialize randomly.
w = torch.randn(Xs.shape[1])
b = torch.randn(1)

# Try it out by predicting on training set.
Xs_tr @ w + b

--> tensor([-428.8886, -384.7284, -212.2810,  ..., -286.6152, -326.9498, -162.2029])
```

Defining a network: The tedious way

- OK, but that just looks like NumPy with extra steps.
- The real magic is in the construction of the computational graph.
- And what we can do with the computational graph.

```
# A quadratic loss function.
def l2loss(y, y_hat):
    return ((y - y_hat) ** 2.0).mean()

# Define our model parameters, initialize randomly.
w = torch.randn(Xs.shape[1], requires_grad=True)
b = torch.randn(1, requires_grad=True)

# Compute the loss on the training set.
l2loss(ys_tr, Xs_tr @ w + b)

--> tensor(909343.6875, grad_fn=<MeanBackward0>)
```

Defining a network: The tedious way

- Our computational graph is instrumented to compute gradients!
- Gradients of the loss wrt tensors with **`requires_grad=True`** will be computed:

```
# Compute the gradient of the loss function.
loss = l2loss(ys_tr, Xs_tr @ w + b)
loss.backward()
print(f'Gradient wrt w: {w.grad}')
print(f'Gradient wrt b: {b.grad}')

-->
Gradient wrt w: tensor([-6.7520e+03, -4.5452e+04, -9.1817e+03, -1.8607e+03,
                        -4.0045e+06, -6.9095e+03, -6.1650e+04,  2.0769e+05])
Gradient wrt b: tensor([-1739.1476])
```

Defining a network: The tedious way

- We could perform gradient descent, but everything must be done just right:

```
# Define our model parameters, initialize randomly.
w = torch.normal(0.0, np.sqrt(1.0/((8+1)/2)), size=(Xs.shape[1],)).requires_grad_()
b = torch.tensor(1.0, requires_grad=True)

# Standardize dataset.
Xs_tr -= Xs_tr.mean(0)
Xs_tr /= Xs_tr.std(0)
```

Defining a network: The tedious way

- We could perform gradient descent, but everything must be done just right:

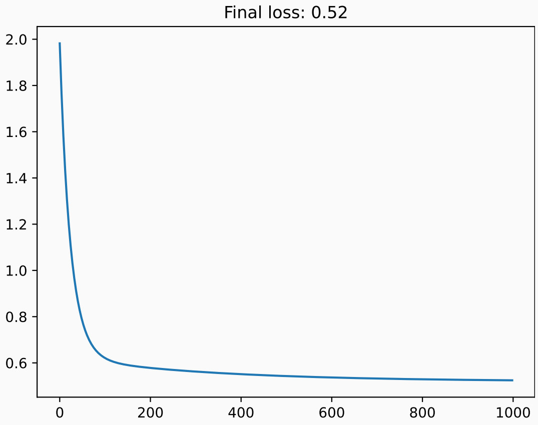
```
# Training hyperparameters.
epochs = 1000
lr = 1e-2

# Training loop.
losses = []
for it in range(epochs):
    loss = l2loss(ys_tr, Xs_tr @ w + b)
    loss.backward()
    w.data -= lr * w.grad.data
    b.data -= lr * b.grad.data
    w.grad.data.zero_()
    b.grad.data.zero_()
    losses.append(loss.item())

# Plot losses and report last loss (MSE).
plt.plot(losses[1:]); plt.title(f'Final loss: {losses[-1]}')
```

Defining a network: The tedious way

- We could perform gradient descent, but everything must be done just right:



Defining a network: The modular way

- OK, that really sucks. And is error-prone. And doesn't scale. And has lots of other problems.
- Instead, we will make use of the reusable modules from the `torch.nn` package.
- This PyTorch namespace contains implementations of very many types of layers. Let's take a look...
- The simplest way to define out linear regression model is:

```
model = nn.Linear(8, 1)
print(list(model.parameters()))
--> [Parameter containing:
    tensor([[-0.1241,  0.2113, -0.1457, -0.2680, -0.3201, -0.2886, -0.2482, -0.0639]],
           requires_grad=True),
    Parameter containing: tensor([0.3110], requires_grad=True)]
```

Defining a network: The modular way

- The classes from the `torch.nn` package encapsulate all parameters and provide a uniform API to access them.
- Their instances act like functions:

```
# Our linear model.
model = nn.Linear(8, 1)

# Training loop.
losses = []
for it in range(epochs):
    loss = l2loss(ys_tr, model(Xs_tr).flatten())
    model.zero_grad()
    loss.backward()
    for p in model.parameters():
        p.data -= lr * p.grad.data
    losses.append(loss.item())
```

Defining a network: The modular way

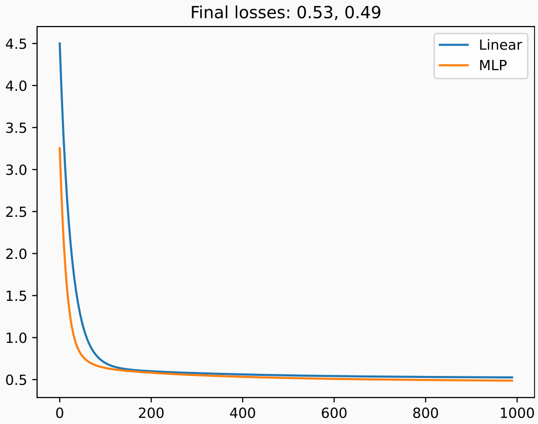
- The modularity really shines when layers are composed:

```
# Our MLP model.
model = nn.Sequential(nn.Linear(8, 16), nn.Tanh(), nn.Linear(16, 1))

# Training loop.
losses_mod = []
for it in range(epochs):
    loss = l2loss(ys_tr, model(Xs_tr).flatten())
    model.zero_grad()
    loss.backward()
    for p in model.parameters():
        p.data -= lr * p.grad.data
    losses_mod.append(loss.item())
```

Defining a network: The modular way

- Which makes model selection reasonably agile:



Defining a network: The modular way

- The most general way to define models is to extend `nn.Module`.
- This way we have complete access to everything and control over composition:

```
class LinRegMLP(nn.Module):
    def __init__(self, num_in, num_hidden, activation=F.tanh):
        super().__init__()
        self.activation = activation
        self.fc1 = nn.Linear(num_in, num_hidden)
        self.out = nn.Linear(num_hidden, 1)

    def forward(self, xs):
        return self.out(self.activation(self.fc1(xs)))
```

Dataset and DataLoader: Basics

- Data is at the heart of Deep Learning.
- The important breakthroughs in Deep Learning would not have been possibile without massive datasets.
- Annotated or unlabeled data may power training of Deep Models.
- Model architectures are strongly related to the type of data.
- Dataset size and annotation quality will determine how large of a model we can train.

Dataset and DataLoader: Batching

- In the examples above we are performing batch gradient descent.
- Usually this is inadvisable, so we use a `DataLoader` which is a batched iterator over a dataset:

```
from torch.utils.data import DataLoader, TensorDataset

# Create a TensorDataset
ds_train = TensorDataset(Xs_tr, ys_tr)

# Create a DataLoader
dl_train = DataLoader(ds_train, batch_size=512, shuffle=True)

for (Xs, ys) in dl_train:
    print(f'Xs: {Xs.shape}, ys: {ys.shape}')
-->
Xs: torch.Size([512, 8]), ys: torch.Size([512])
Xs: torch.Size([512, 8]), ys: torch.Size([512])
...
Xs: torch.Size([120, 8]), ys: torch.Size([120])
```

Dataset and DataLoader: The rest of the story

- Datasets for Deep Learning tend to be massive, and data loading is a critical issue.
- The `DataLoader` might be responsible for loading data from disk, transforming it (e.g. standardization or data augmentation), and batching it for training and validation.

Optimization: Training Deep Models

- Without optimization we cannot get anything out of data.
- There are a variety of optimization algorithms that manage training of Deep networks.
- 99% of them are based on gradient-based optimization (i.e. Stochastic Gradient Descent).
- There is also a (rather large) bag of tricks needed to avoid common training pitfalls.

Optimization: Training Deep Models

- The `torch.nn` module provides classes to encapsulate details of model definitions.
- In the `torch.optim` package you will find implementations of several optimization algorithms the encapsulate the details of model training:
    - `torch.optim.SGD`: Implements the standard Stochastic Gradient Descent (SGD) algorithm, with optional momentum.
    - `torch.optim.Adam`: Implements the Adaptive Moment Estimation (Adam) algorithm, which augments SGD with adaptive, per-parameter learning rates.

Optimization: Putting Everything Together

- First some imports and a model definition:

```
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD, Adam
from tqdm.notebook import tqdm

# Define our model.
class LinRegMLP(nn.Module):
    def __init__(self, num_in, num_hidden, activation=F.tanh):
        super().__init__()
        self.activation = activation
        self.fc1 = nn.Linear(num_in, num_hidden)
        self.out = nn.Linear(num_hidden, 1)

    def forward(self, xs):
        return self.out(self.activation(self.fc1(xs)))
```

Optimization: Putting Everything Together

- And a function to train the model:

```
def train(model, dl_train, dl_val=None, epochs=100, lr=1e-3):
    opt = Adam(model.parameters(), lr=lr) # We'll use Adam.
    (train_losses, val_losses) = ([], [])

    # Iterate for required number of epochs.
    for epoch in tqdm(range(epochs)):
        # Iterate over all batches.
        epoch_loss = 0.0
        for (Xs, ys) in dl_train:
            ys_hat = model(Xs).flatten()  # Forward pass and loss computation.
            loss = F.mse_loss(ys, ys_hat)
            opt.zero_grad()               # Take a gradient step.
            loss.backward()
            opt.step()
            epoch_loss += loss.item()
        train_losses.append(epoch_loss / len(dl_train))  # Collect average epoch loss.

        # If we have a validation dataloader, compute average loss over it.
        if dl_val:
            with torch.no_grad():
                val_losses.append(np.mean([F.mse_loss(ys, model(Xs).flatten()).item() for (Xs, ys) in dl_val]))
    return (train_losses, val_losses)
```

Optimization: Putting Everything Together

- And (finally) we can use this training loop:

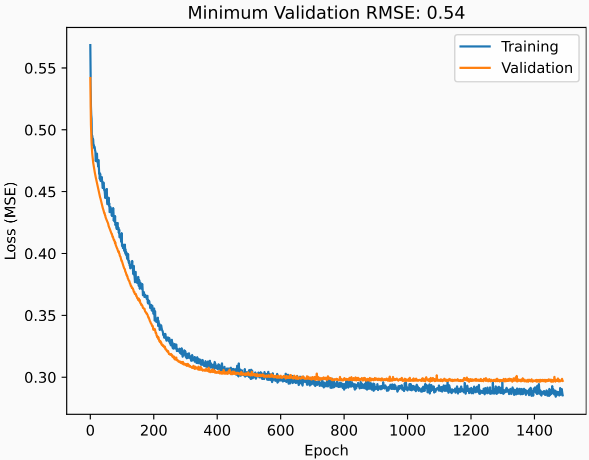
```
# Training hyperparameters.
lr = 1e-3
epochs = 1500
batch_size = 512
inner_size = 16

# Create datasets and DataLoaders.
ds_train = TensorDataset(Xs_tr, ys_tr)
ds_val = TensorDataset(Xs_te, ys_te)
dl_train = DataLoader(ds_train, batch_size=batch_size, shuffle=True)
dl_val = DataLoader(ds_val, batch_size=batch_size, shuffle=False)

# Instantiate model, train it, and plot losses.
model = LinRegMLP(8, inner_size)
(train_losses, val_losses) = train(model, dl_train, dl_val, lr=lr, epochs=epochs)
```

Optimization: Putting Everything Together

- And arrive at something like this:



Discussion

Connectionist models and the state-of-the-art

- Deep Neural Networks are connectionist models that represent the state-of-the-art for many learning problems today.
- They are usually trained with backpropagation.
- Since they typically have *very many parameters* they must be trained with extreme care.
- They also typically require more training data than classical approaches in order to generalize.
- Note that there is nothing fundamentally different about these models with respect to the classical models we have already seen.
- They simply represent a much richer hypothesis set of functions $\mathcal{H}$ over which we search.

The Embarrassment of Choice

- PyTorch is the de facto framework of choice for most Deep Learning projects.
- It is not the only one:
  - Tensorflow/Keras: a graph-based, automatic differentiating framework for deep learning in Python – now with JAX and PyTorch backends!
  - JAX: A Python library for accelerator-oriented array computation and program transformation – sold as differentiable Python programming.
- Moreover, there are many frameworks for tracking experiments:
  - Tensorboard: A suite of web applications for inspecting and understanding your TensorFlow (and PyTorch, and JAX, and Keras, and...) runs and graphs.
  - Weights and Biases: Track, version and visualize with just 5 lines of code, reproduce any model checkpoints, monitor CPU and GPU usage in real time.
  - Comet: Provides an end-to-end model evaluation platform for AI developers, with best in class LLM evaluations, experiment tracking, and production monitoring.

Reading and Homework Assignments

Reading Assignment:

- Bishop: Chapter 5
- PyTorch: This is an excellent introductory tutorial