Homework

• Some typos in Homework 1 solutions! Please check latest file!

• Homework 1 final/corrections due Wednesday 1/25 5 5pm

• Homework 2 to be released Wednesday 1/25 as well

• Draft due Friday 2/3 5pm

• Final/corrections due Wednesday 2/8 5pm

Note: This last part turned out to be more difficult than I anticipated so all answers should get credit.

My initial idea for a solution of two engineered features that would allow the spiral pattern to be linearly separated was $r = \sqrt{x_1^2 + x_2^2}$ and $\theta = \arctan(2(x_1, x_2))$. The four-quadrant $\arctan(2(x_1, x_2))$ has an output range that covers the full $[-\pi, \pi]$ range as defined in https://en.wikipedia.org/wiki/Atan2. I also had to swap $x_1 \leftrightarrow x_2$ from how you would usually define this because of a bug in the TensorFlow Playground code! See https://github.com/tensorflow/playground/blob/02469bd3751764b20486015d4202b792af5362a6/src/dataset.ts#L145-L146.

However, if we plot these two features, we see that the resulting dataset is not linearly separable! After staring at the source code of how the dataset is generated and some trial and error, we can come up with an engineered feature that (together with $r$) lets the dataset be linearly separated, namely $(\theta - 2r) \mod 2\pi$ (and there are many variations possible). See the figure below.

Figure 11: Spiral dataset in $(x_1, x_2)$ (left), in $(r, \theta)$ (center), and in $(r, (\theta - 2r) \mod 2\pi)$ (right).
Recap: (Multiclass) logistic regression

Logistic regression

$y = \text{softmax}(w^T x)$

$\nu_\mu + n \rightarrow \mu + p$ (CC)

Deep learning algorithms have been successful in tasks like image recognition. These networks–and in particular convolutional neural networks (CNNs)–present several advantages with respect to the traditional identification methods described in Section 1. Not only do traditional algorithms rely heavily on the efficiency of the geometric separation of the components, they are also limited in that the features they employ for identification are only those

As seen in Figure 1 the outgoing lepton carries the same flavor as the original neutrino by lepton conservation.
Linear models & embeddings

Data

Linear classifier

Embedding + Linear classifier

\[ y = \text{softmax}(w^T x) \]

\[ y = \text{softmax}(w^T \phi(x)) \]

We have seen the polynomial embedding:

\[ \phi(x) = (1, x, x^2, \ldots, x^n) \]
Limitations of linear models

- **Problem**: A linear model considers each feature $x^{(i)}$ independently and regresses the weight $w^{(i)}$ with which it contributes to the label.

- But often individual low-level features (e.g., pixels in an image) are not meaningful. What matters is the relationship between pixels.

- **Example**: To recognize a $\nu_\mu + n \rightarrow \mu + p$ interaction, we need to look at parts and the relationship between parts.

In theory, we can learn an embedding $\phi(x)$ that encodes all this (and this is done), but can we learn it?
One artificial neuron

input features

weights

sums

non-linearities

\( x_1 \)

\( x_2 \)

\( x_M \)
Two artificial neurons

- Input features: \( x_1, x_2, \ldots, x_M \)
- Weights: \( \mathbf{w} \)
- Sums: \( \sum \)
- Non-linearities: \( f \)
- Output features:

\[
\begin{align*}
\sum (x_1w_1 + x_2w_2 + \cdots + x_Mw_M) \rightarrow f \rightarrow \text{output features}
\end{align*}
\]
$N$ artificial neurons
$N$ artificial neurons form a layer
$N$ artificial neurons form a layer
$N$ artificial neurons form a layer
Multiple layers form a network
One artificial neuron: weighted sum and non-linearity

\[ s = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^\top \mathbf{x} \]

\[ h = \sigma(s) \]
One artificial neuron: weighted sum and non-linearity

\[
\text{input features} \quad \rightarrow \quad \text{sum} \quad \rightarrow \quad \text{non-linearity} \quad \rightarrow \quad \text{output feature}
\]

\[\frac{1}{\mathbf{w}} \cdot \mathbf{x} \overbrace{\sigma} = \sigma(\text{sum of weighted features})\]
$N$ artificial neurons in a layer

Layer: parallelized weighted sum and non-linearity

\[ s_j = w_j^T x \quad \xrightarrow{\text{one sum per weight vector}} \quad s = W^T x \quad \xrightarrow{\text{vector of sums from weight matrix}} \quad h = \sigma(s) \]
artificial neurons in a layer

\[ h = (s) \]

layer: parallelized weighted sum and non-linearity

\[ \sum \text{sums} \quad \sum \text{non-linearities} \]

\[ = \]

\[ \sigma(\sum \text{sums}) \]

\[ = \]

\[ \sigma(\sum \text{weights}) \]

\[ \sigma(\sum \text{non-linearities}) \]

\[ \text{input features} \]

\[ \text{output feature} \]

\[ N \]
**Layers in a network**

A network is a sequence of parallelized weighted sums and non-linearities. Define $x^{(0)} \equiv x, \ x^{(1)} \equiv h, \ ETC.$

**1st layer**

$$s^{(1)} = \mathbf{W}^{(1)\top} x^{(0)}$$

$$x^{(1)} = \sigma(s^{(1)})$$

**2nd layer**

$$s^{(2)} = \mathbf{W}^{(2)\top} x^{(1)}$$

$$x^{(2)} = \sigma(s^{(2)})$$

...
Layers in a network

**Network**: sequence of parallelized weighted sums and non-linearities

\[ \underbrace{\sigma(\ldots\sigma(\sigma( \underbrace{\sigma(\ldots\sigma(}^{\text{1st weights}}( \underbrace{\sigma(\ldots}^{\text{2nd weights}}( \underbrace{\ldots)}^{\text{output}}) \ldots)) \ldots)) \ldots) }_{\text{input}} \]
Role of nonlinearities

If we didn’t have non-linearities, the whole network would reduce to a linear function!
Nonlinearities and coordinate changes

The dot product is the distance between a point and a plane.

Each artificial neuron defines a (hyper)plane:

$$0 = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_M x_M$$

Calculating the weighted sum corresponds to finding the shortest distance between the input point and the weight hyperplane.

Distance from hyperplane (up to a factor)
Reinterpretation

the non-linearity transforms this distance, creating a field that changes non-linearly with distance
Neural networks & topology

Data

Linear classifier

2-layer network

$$y = \text{softmax}(wx)$$

$$y = \sigma(W_2\sigma(W_1x))$$

Neural networks & topology

colah.github.io/posts/2014-03-NN-Manifolds-Topology
**NNs are universal function approximators**

Universal approximation theorem (informal). Given a function $y = f(x)$ and an $\epsilon > 0$, there exists a deep network $y = f_w(x)$ (of arbitrary width or depth) such that:

$$\sup_{x \in X} ||f(x) - f_w(x)|| < \epsilon$$

Note: This means that a network can *represent* any function, not that it can learn it! The “amount” of function a given network can represent is often called its expressive power.
Training a NN

We train deep networks using **Maximum Likelihood Estimation (MLE)**: The last layer of a DNN is a softmax that outputs probabilities over classes:

\[
p_w(y|x) = \begin{bmatrix} 0.9 \\ 0.1 \\ \vdots \end{bmatrix}
\]

- \( x = \text{input data} \)
- \( w = \text{vector containing all weights} \)
- \( y = \text{label} \)

We train the weights \( w \) to maximize the log-likelihood of the data under our model:

\[
L(w) = -\frac{1}{N} \sum_{i=1}^{N} \log p_w(y_i|x_i)
\]

**Negative log-likelihood loss (cross-entropy loss)**
Gradient descent

- Start from some initial value $w_0$ of the parameters
- For $t = 0, 1, 2, \ldots$ do the following:
  - Compute the gradient $\nabla_w L(w_t)$ (direction of steepest increase of $L(w)$ at $w_t$)
  - Take a small step in the opposite direction: $w_{t+1} = w_t - \eta \nabla_w L(w_t)$

Problem: Deep networks have millions or billions of weights. We can can’t naïvely compute all gradients independently!
Backpropagation (i.e. the chain rule)

\[ y = f_L(\ldots f_2(f_1(x)))\ldots \]

\[ y = f_L(x_L) \]
\[ x_L = f_{L-1}(x_{L-1}) \]
\[ \vdots \]
\[ x_1 = f_1(x) \]

We want to compute \( \frac{\partial y}{\partial x_l} \) for all \( l \in \{1,\ldots,L\} \)

At each step we can reuse the computation of the previous step!
Backpropagation for NNs

\[ \frac{\partial \mathcal{L}}{\partial \mathbf{W}(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}(L)} \frac{\partial \mathbf{x}(L)}{\partial \mathbf{s}(L)} \frac{\partial \mathbf{s}(L)}{\partial \mathbf{W}(L)} \]

- depends on the form of the loss
- derivative of the non-linearity

\[ \frac{\partial}{\partial \mathbf{W}(L)} (\mathbf{W}(L)^\top \mathbf{x}^{(L-1)}) = \mathbf{x}^{(L-1)^\top} \]

Note \( \nabla_{\mathbf{W}(L)} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}(L)} \) is notational convention.
Backpropagation for NNs

now let’s go back one more layer…

again we’ll draw the dependency graph:

\[
\frac{\partial L}{\partial W(L)} = \frac{\partial L}{\partial x(L)} \frac{\partial x(L)}{\partial s(L)} \frac{\partial s(L)}{\partial W(L)}
\]

Note: we can reuse previous calculations!
Backpropagation for NNs

- We can use backpropagation to compute the gradients on any computation graph

\[ y = \sin(w_1x + \log(x)) + \cos(x) \]

- Modern deep neural networks can have a very complex structure!
Automatic differentiation

we need to manually implement backpropagation and weight updates

→ can be difficult for arbitrary, large computation graphs

most deep learning software libraries automatically handle this for you

TensorFlow  
PyTorch  
mxnet  
Keras  
Caffe2  

and many more

just build the computational graph and define the loss
Implementation & training issues
NNs are highly parallelizable

**unit parallelization**
perform all operations within a layer simultaneously

**data parallelization**
process multiple data examples simultaneously

"batch" of data well suited for mini-batch SGD
NNs and GPUs

- Single instruction multiple data (SIMD)
Vanishing gradients

difficult to train very deep networks with saturating non-linearities

saturating non-linearities have small derivatives almost everywhere

in backprop, the product of many small terms (i.e. \( \frac{\partial x^{(\ell)}}{\partial s^{(\ell)}} \)) goes to zero

\[
\frac{\partial L}{\partial W^{(\ell)}} = \cdots \frac{\partial x^{(L)}}{\partial s^{(L)}} \cdot \frac{\partial x^{(L-1)}}{\partial s^{(L-1)}} \cdot \frac{\partial x^{(\ell+1)}}{\partial s^{(\ell+1)}} \cdot \frac{\partial x^{(\ell)}}{\partial s^{(\ell)}} \cdot \frac{\partial s^{(\ell)}}{\partial W^{(\ell)}}
\]

difficult to train very deep networks with saturating non-linearities
Nonlinearities

**saturating**

*derivative goes to zero at $+\infty$ and $-\infty$*

**non-saturating**

*non-zero derivative at $+\infty$ and/or $-\infty$*

**hyperbolic tangent** (tanh)

**rectified linear unit** (ReLU)

**leaky ReLU**

**softplus**

**exponential linear unit** (ELU)

**most often used**

$\text{ReLU}(x) := \max(0, x)$
Weight initialization

Initialize the weights so that if the input $x_l$ to the $l$-th layer has variance $\text{var}(x_l) = 1$ then the output $x_{l+1} = \text{ReLU}(W_l \cdot x_l)$ also has $\text{var}(x_{l+1}) = 1$.

Kaiming Initialization:

$$w_l \sim \mathcal{N}(0, 2/\text{dim}(x_l))$$

sample the weights from a gaussian distribution with variance inversely proportional to the size of the layer input.
Batch normalization

→ keep the inputs within the dynamic range of the non-linearity

we can \textit{normalize} the activations before applying the non-linearity

\[
s \leftarrow \frac{s - \text{shift}}{\text{scale}}
\]
Why does batch normalization work?

**original motivation:** *internal covariate shift*

changing weights during training results in changing outputs; input to the next layer changes, making it difficult to learn

batch norm. should stabilize the activations during training
Why does batch normalization work?

*but actually...*

batch norm. does *not* seem to significantly reduce internal covariate shift

rather, it seems that batch norm. stabilizes and smooths the optimization surface

(topic of ongoing research)

---

*How Does Batch Normalization Help Optimization?*, Santurkar et al., 2018
Optimizing nonconvex functions
Loss landscape of NNs

Convex problem
(logistic regression, SVMs)

Deep Networks
Consequences of nonconvexity

Sensitivity to initialization: based on where you start you may end up in different minima

Shallow minima: we may get stuck in a suboptimal local minimum

But SGD can jump out!

The noise of stochastic gradient descent is actually a benefit in deep learning!
Flat & sharp minima

To converge to a minimum we need:

$$\eta < \frac{2}{\text{curvature}}$$

The noise of SGD makes us jump out of sharp minima

If the learning rate is too large we can’t enter a sharp minimum

Is this a problem? In deep learning it is often observed that flat minima are better solutions, so avoiding sharp minima is good!
Learning rate annealing

We start with a high learning rate. Converges faster and avoids sharp minima. But to converge we need to decrease the learning rate later. If we decrease too fast we end up in a bad minimum, so we do it in multiple steps.
Residual connections

sequential connectivity: *information must flow through the entire sequence to reach the output*

- Without residual connections
- With residual connections

information may not be able to propagate easily

*make shorter paths to output*

residual & highway connections

dense (concatenated) connections

*Deep residual learning for image recognition, He et al., 2016*

*Highway networks, Srivastava et al., 2015*

*Densely connected convolutional networks, Huang et al., 2017*

Pictures from https://arxiv.org/abs/1712.09913
Generalization
Data memorization

Given a training dataset with millions of completely random labels, DNNs networks can easily reach zero training error. They do so by memorizing the association between meaningless but unique patterns in the samples and the label.

The problem is that they learn these degenerate patterns even on real data… (which is also a privacy risk)
Generalization bounds

One can show that the “generalization gap” is bounded by the amount of information memorized by the network:

\[ L_{\text{test}} - L_{\text{train}} \leq \sqrt{\frac{I(w; D)}{N}} \]

Information that the weights contain about the training examples

Ways to limit the information stored in the weights:

- **stochasticity (uncertainty)**: batch norm, SGD, dropout
- **constraints**: early stopping, weight penalties
Next time

- Optimizers
- Training issues
- Optimization tips and tricks
- Keras hands-on exercise