CS337: Artificial Intelligence and Machine Learning

Krishna Agaram

Autumn 2023

Contents

Chapter 1

The neural network

An interesting application: smelling with NNs.

1.1 The Mc-Culloch-Pitts Neuron model

This is one of the simplest model of an object that learns from data. McCulloch and Pitts proposed this model in 1943, calling it an artificial neuron (we will henceforth simply call it a neuron). Data $x = (x_1, \ldots, x_n)$ is passed to the neuron, which has weights $w = (w_1, \ldots, w_n) \in \mathbb{R}^n, b \in \mathbb{R}$. The neuron computes the quantity $f_{w,b}(x) = w^T x + b$, where $f_{w,b}$ called the aggregate function. The neuron then passes the result through a function g called the activation function. The output of the neuron is defined to be $g(f_{w,b}(x))$. Note that the activation of a neuron on input x is simply its output on input x.

Remark. Consider the case where g is the sign function. This neuron model is precisely the perceptron model! Indeed, in 1957, Rosenblatt proposed the perceptron model inspired by the McCulloch-Pitts neuron model. And thus, one neuron can be used to perform binary classification.

1.2 History of deep networks

- 1943: McCulloch and Pitts propose the McCulloch-Pitts neuron model.
- 1957: Rosenblatt proposes the perceptron model.
- 1960: The idea of backpropagation is proposed by a controls engineer, Kelley. It was actually developed for flight path control.
- 1969: Minsky and Papert show that the perceptron model cannot learn the XOR function.
- 1986: Rumelhart, Hinton, and Williams show that backpropagation can be used to train neural networks with hidden layers.
- 1989: The convolutional neural network is proposed by LeCun.
- 1997: The long short-term memory (LSTM) model is proposed by Hochreiter and Schmidhuber.
- 2009: Deep learning for speech recognition is proposed by Dahl.
- 2012: AlexNet wins the ImageNet competition.
- 2014: Generative adversarial networks (GANs) are proposed by Goodfellow.

- 2016: AlphaGo defeats Go world champion Lee Sedol.

The advent of deep (many hidden layers) neural networks has been made possible thanks to the following factors:

1. Vast amounts of data
2. Faster computation: specialized hardware like GPUs (and recently TPUs) for matrix computation
3. Better algorithms: better initialization, better activation functions, better optimization techniques, toolkits, libraries like TensorFlow, PyTorch, etc.

1.3 The multi-layer perceptron

Okay, back to the neuron. Typically, each neuron is expected to 'do its bit', that is, capture something interesting about the data, some pattern, etc. Suppose we have d neurons, all receiving the same input x, but having different weights and activations. They all capture different things in the data, that is: two inputs may appear quite similar (the outputs corresponding to both those inputs are close) to one neuron, but they may appear to be quite different to another! That is, different weights capture different separations of the data. Suppose we could use these varied outputs in some way. Here is the key idea:

The output of the d neurons can be passed as the input to another neuron, to form a more complicated neural network. On input x, this 'second-level' neuron now has different details about the input x fed to it by all the 'first-layer' neurons. This is shown in the diagram below.

How does the final output $\hat{y}$ look? $\hat{y} = g_2(w^T x_1 + b)$, where x_1 is the vector comprising the outputs of the first layer of neurons. That is, $(x_1)_i = g_1(w_i^T x + b_i)$, where w_i is the weight vector of the ith neuron in the first layer, and b_i is the bias of the ith neuron in the first layer. If we suppose that the activations of the first layer are all identical (doesn't hurt performance, but allows vectorization which greatly speeds up training), we can write this in vectorized form as $x_1 = g_1(W_1 x + b_1)$, where W_1 is the matrix whose ith column is w_i, and b_1 is the vector whose ith entry is b_i. Thus, we can write:

$$\begin{aligned} x_0 &= x \\ x_1 &= g_1(W_1 x_0 + b_1) \\ \hat{y} = x_2 &= g_2(W_2 x_1 + b_2) \end{aligned}$$

This would serve quite well for regression where we need a scalar output. Suppose instead that we now have a classification problem, say classify an input x into one of the classes $0, 1, \ldots, 9$. A simple way to use a network for this is to have 10 *second-layer* neurons, and then predict the class to be the one corresponding to the second-layer (also called output layer) neuron with the highest output.

Key Idea 2

We can use the outputs of the second layer to make a prediction for classification/regression.

But why stop here? Perhaps we can use the outputs of the second layer to feed into a third layer, and so on. This is the idea behind feedforward deep neural networks, also called multi-layer perceptrons (MLPs).

Key Idea 3

We can stack multiple layers of neurons next to each other to form a deep neural network.

1.4 The loss function

Before we get too excited about deep neural networks, we need to figure out how to train them. That is, we need to figure out how to choose the weights and biases of the neurons so that the network does what we want it to do. To do this, as before, we need to define a loss function to tell the network how well it is doing. The loss function is a function of the weights and biases of the network, and measures how well the network is doing. The goal is to minimize the loss function.

1.4.1 Typical loss functions

For regression problems (the output vector of the MLP is a scalar, aka there is only one output neuron), we typically use the squared error loss. Suppose the true output is y, and the output of the MLP is $\hat{y}$. Then, the squared error loss is defined as:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

For classification problems, the cross-entropy loss is typically used. Suppose we have k classes, and the output vector of the MLP is k-dimensional. Let y be the true label of the input x, and let $\hat{y}$ be the output of the MLP on input x. Then, the cross-entropy loss is defined as:

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

What do we use from the neural network outputs to describe the probabilities $\hat{y}_k = f(y = k|x)$? We had used the sigmoid for binary classification. Here we use a generalization, called the softmax function. The softmax function takes a k-dimensional vector z and outputs a k-dimensional vector $\sigma(z)$, where:

$$f(y = i|x) = \frac{e^{o_i}}{\sum_{j=1}^{k} e^{o_j}}$$

where $(o_1, \ldots, o_k)$ is the output of the network on input x.

1.4.2 Common activation functions

Suppose the activation function of every neuron in the network was linear. Then the final output vector is just some linear function of the input vector, since composing linear functions gives a linear function. This is not very useful, since we want the network to be able to capture non-linearities in the data. Plus, it is then just a complicated way of doing linear regression. But hang on, see this.

Thus, we need to use non-linear activation functions. Here are some common ones:

1. **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$. A common choice for the activation function. It is smooth, and bounded. However, it suffers from the vanishing gradient problem: the gradient of the sigmoid function is very small for large values of x, and so the weights corresponding to neurons with large activations do not get updated much. This is a problem because the neurons with large activations are the ones that are most important in the network. This slows down learning significantly.

2. **Hyperbolic tangent**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This is similar to the sigmoid function, but is zero-centered. This helps with learning, since the gradient is not always positive or always negative. However, it still suffers from the vanishing gradient problem.

3. **ReLU**. A very popular choice, and converges quickly.

4. **Many more!**: Leaky ReLU, ELU, SELU, etc.

Chapter 2

Backpropagation

2.1 Training the network

Now to the real deal. How do we get this to learn what the right weights are? As a function of the weights, the loss is complicated and not convex. We can still try to use gradient descent, but we have to be careful about falling into flat regions and local minima of the loss function. For simplicity, suppose we use plain old SGD:

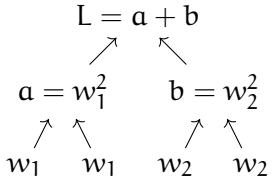
Algorithm 1 SGD for training a neural network

1: Initialize weights and biases randomly
2: **while** not converged **do**
3: Sample a minibatch of data $\mathcal{D}$
4: Compute the gradient of the loss function $\nabla_w(L)$ on $\mathcal{D}$ with respect to the weights
5: Update the weights and biases using the gradient. Set $w \leftarrow w - \alpha\nabla_w(L)$
6: **end while**

We now address the elephant in the room: how on earth do you compute the derivative of the loss with respect to each weight?!

2.2 Backpropagation

Set the loss function to be minimized as a loss L. We can write the dependence of the loss on the weights as a directed acyclic graph on the weights. The nodes are the weights, and the edges are the dependence of the loss on the weights. For example, if the loss is $L = L(w_1, w_2) = w_1^2 + w_2^2$, then the graph is:



Suppose that the loss depended on a weight w by $\mathcal{L} = g(u)$, where $u = f(x)$. We have, by the chain rule, the expression

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w}$$

In our case above, $g(a) = a + b$, $a = w_1^2$. We get

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_1} = 1 \cdot 2w_1$$

. Similarly, $\frac{\partial L}{\partial w_2} = 2w_2$. And we're done. In a more complicated example, how will this work? We need the multuvariable chain rule.

Theorem 2.1 (Multivariable chain rule). *Suppose* $f = f(y_1, \ldots, y_n)$ *is a function. For some* i*, suppose* $y_i = g(x_i)$*. Then*

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

In our case, $f = L$ and the y_is are the immediate children of the loss in the DAG. x_i is a weight (which is at the leaf of the DAG).

The theorem above tells us that we can compute $\frac{\partial L}{\partial w}$ if we know $\frac{\partial u}{\partial w}$ for every child u of L. The relations $\frac{\partial L}{\partial u}$ are local, and are simply functions of the children of L. (For example, if $L(u_1, \ldots, u_k) = u_1^2 + u_2 * \tanh u_3$, then $\frac{\partial L}{\partial u_1} = 2u_1$, $\frac{\partial L}{\partial u_2} = \tanh u_3$, $\frac{\partial L}{\partial u_3} = u_2/\cosh^2 u_3$.)

This is a recursive procedure, and thus we can compute the derivatives by starting at the leaves and working our way up to the root. This, in essence, is backpropagation.

2.3 The algorithm

The algorithm involves two passes: a forward pass and a backward pass. The forward pass computes the values of the nodes in the DAG. The backward pass computes the derivatives of the loss with respect to the weights (and thus we have the gradient).

2.3.1 Forward pass

The forward pass is simple. We simply compute the values of the nodes in the DAG in their topological order.

Algorithm 2 forward pass

1: The values of the leaves are the input values and the values of the weights.
2: **for** each node u in the DAG in topological order **do**
3: compute $u.val = f(u.child_1.val, u.child_2.val, \ldots, u.child_k.val)$
4: **end for**

2.3.2 Backward pass

Here we use the computed values to compute the gradient of every node with respect to the leaves that vary (i.e. the gradient of a node wrt the input is 0, and we do not bother about these).

While building the DAG itself, we store the dependences of the parent $di = \frac{\partial u}{\partial(u.child_i)}$ on its children as **functions**. These depend simply on the children's values. Note that we use some of the input's values as required (for example, in $w^T x$) in the functions di but we do not include them as children because they are not variables that we are differentiating with respect to.

Then the algorithm proceeds recursively like so:

Algorithm 3 backward pass

1: The function dw for the leaves (weights) is set to 1.
2: **for** each node u in the DAG in topological order **do**
3: compute $u.grad = \sum_{i=1}^{k} u.di \times child_i.grad$ (where $\star.grad$ is a vector with dimension equal to the number of weights)
4: **end for**

2.4 Vectorized implementation of backpropagation

2.5 Regularization

2.5.1 L2 regularization

Introduce another loss term that penalizes the squared magnitude of **all** parameters. That is, for every weight w, add the term λw^2 to the loss. This is called L2 regularization. The intuition, as before, is that this will penalize large weights, and thus prevent overfitting.

2.5.2 Dropout

During training, randomly set the activations of some neurons to 0. This is called dropout. This prevents the network from relying too much on any one neuron, and thus prevents overfitting. Theoretical guarantees make use of the fact that the neural network is an average of exponentially many smaller networks.

We do not dropout any of the input or output neurons, of course. An important point: at test time, we do not use dropout. Instead, we multiply the activations of each neuron by the probability that it was not dropped out during training - the expected value of the activation is used.

Remark. An extreme version of dropout is to randomly drop out entire layers of neurons. This is called dropconnect. This makes use of residual connections, which connect the input of a layer to the output of a layer that is not the immediate previous layer.

2.5.3 Early stopping

Stop training when the validation loss starts increasing. This is called early stopping. This is a form of regularization because it prevents the network from overfitting to the training data (we stop training before it can do so).

Chapter 3

Learning Rate Scheduling

Is vanilla SGD the best way to train a neural network? What is a *good* learning rate? Figure 3.1 shows the effect of the learning rate on the loss function.

3.1 Momentum

There are two main motivations for momentum, which are limitations of SGD:

- SGD can have large weight updates on complex loss landscapes. The gradient is all over the place and the weight updates make training unstable.
- At flat points like saddle points, SGD can get stuck. The gradient is zero, and the weight updates are zero. How do we get out of a flat point?

SGD with momentum addresses these issues. Consider $0 \leq \beta < 1$ and define a velocity term v_t as follows:

$$v_t = \beta v_{t-1} + \eta(\nabla_w L)(w_t)$$

That is, the velocity is an exponentially decaying average of the gradients:

$$\begin{aligned} v_t &= \eta(\nabla_w L)(w_t) + \beta v_{t-1} \\ &= \eta(\nabla_w L)(w_t) + \beta\eta(\nabla_w L)(w_{t-1}) + \beta^2 v_{t-2} \\ &= \eta \sum_{i=0}^{t} \beta^i (\nabla_w L)(w_{t-i}) \end{aligned}$$

The weight update is then:

$$w_{t+1} = w_t - v_t$$

The parameter β is a measure of how much we value old gradients. A value of 0 results in SGD, and a value close to 1 results in a very smooth (but not very relevant to the current step) gradient.

Notice how this allows us to escape flat points. The only issue, is that it tries to escape minima as well! Since gradients are small near a minima, the velocity gradually decreases, but it is true that one will overshoot first and then come back. The velocity will likely be small when it comes back, and so it will not overshoot again.

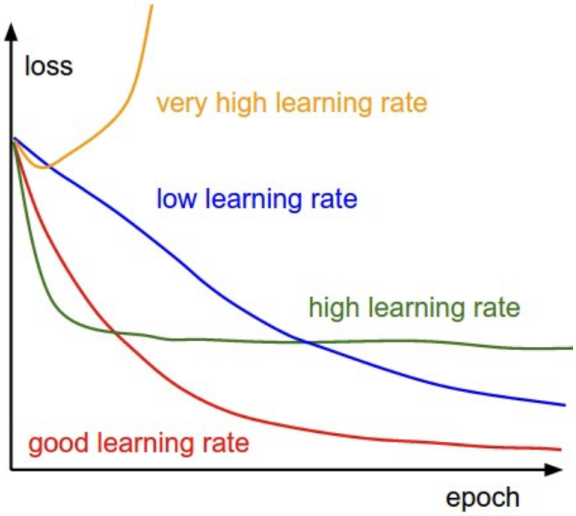


Figure 3.1: What is a good learning rate?

3.2 ADAgrad

The idea behind ADAgrad (adaptive gradient) is to have a different learning rate for each parameter - learning rates adapt to different parameters. The motivation is that some parameters are more sensitive than others, and so we should update them more slowly. Define the operator $\odot$ to be elementwise multiplication. We maintain the vector s_t defined by

$$s_t = s_{t-1} + g_t \odot g_t$$

where $g_t = \nabla_w L(w_t)$ is the gradient at time t. The weight update is then:

$$w_{t+1} = w_t - \eta \frac{1}{\sqrt{s_t} + \epsilon} \odot g_t$$

where ϵ is a small constant to avoid division by zero.

Putting it in one line, the learning rate for the ith parameter is:

$$\eta_i = \frac{\eta}{\sqrt{\sum_{t=1}^{T} (\nabla_w L(w_t))_i^2} + \epsilon}$$

In some sense this is a normalization of the gradient. The weight update is then:

$$(w_{t+1})_i = (w_t)_i - \eta_i((\nabla_w L)(w_t))_i$$

Remark. This is an algorithm that is backed by theory. See Duchi et al. (2011) for more details.

There is an obvious problem with this algorithm. The denominator is a sum of squares, and so it keeps increasing. This means that the learning rate keeps decreasing, and so the algorithm will eventually stop learning. This is solved by the next algorithm.

3.3 RMSProp

We continue to maintain vector s_t, except

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t \odot g_t$$

The update step is identical. The decay allows us to forget old gradients, and so the learning rate does not necessarily keep decreasing.

Remark. This was not a paper! This was actually proposed in a Coursera course by Geoff Hinton.

3.4 ADAM

The best of both worlds. We maintain two vectors m_t and s_t defined by

$$v_t = \beta v_{t-1} + (1-\beta) g_t$$
$$s_t = \gamma s_{t-1} + (1-\gamma) g_t \odot g_t$$

where $g_t = \nabla_w L(w_t)$. The update step is then:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t} + \epsilon} \odot v_t$$

The actual paper does something slightly different: the weight update is

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \odot \hat{v}_t$$

where

$$\hat{v}_t = \frac{v_t}{1-\beta^t}$$
$$\hat{s}_t = \frac{s_t}{1-\gamma^t}$$

This is to correct for the fact that the vectors v_t and s_t are initialized to zero, and so they are biased towards zero. Okay, so what?

We would like to keep v_t always a moving average of gradients. We have:

$$v_t = (1-\beta)\left[\sum_{i=0}^{t-1} \beta^i g_{t-i}\right]$$

with sum of coefficients being $(1-\beta^t)$. For t large, this is not a problem. But it does help to divide by $(1-\beta^t)$ to make the velocity a moving average of gradients.

Chapter 4

Convolutional Neural Networks

Consider the problem of recognizing an object in an image. Humans are able to do this very well, but how do we get a computer to do this? We would like the following properties from a model that learns to do this:

- **Translation invariance**. If we move the object around in the image, we would like the model to still recognize it.
- **Locality**. ?
- **Parameter-efficiency**. We would like to learn as few parameters as possible.

A standard feedforward network where the input is the flattened image ticks none of these boxes. Neural networks with a twist - convolutions are required. From this point on, we shall continue to modify the vanilla neural network for various tasks.

4.1 Cross-correlation

Definition 4.1. Let x and w be vectors of length n. The **cross-correlation** of x and w is defined as:

$$(x \star w)_i = \sum_{j=1}^{n} x_{i+j-1} w_j$$

Chapter 5

Recurrent Neural Networks

There is a paper that finds Automata that are equivalent to RNNs, authors: Weiss et al. 2018. Name of the paper: Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples.

Chapter 6

Unsupervised Learning: Clustering

Consider a set of points $X = \{x_1, \ldots, x_n\}$ in $\mathbb{R}^d$. We want to find similar (defined by close-by according to some metric on $\mathbb{R}^d$) points and group them together (assign them the same class). This is called *clustering* (into classes).

Definition 6.1 (Clustering). Let $X = \{x_1, \ldots, x_n\}$ be a set of points in $\mathbb{R}^d$. A *clustering* of X is a partition of X into k subsets $C_1, \ldots, C_K$ such that - in some well-defined sense - points in the same subset are similar and points in different subsets are dissimilar.

Of what utility is clustering? Consider the following applications:

1. *Market segmentation*: Given a set of customers, group them into clusters based on their purchasing behavior.
2. *Social network analysis*: Given a set of users, group them into clusters based on their social interactions.
3. *Image segmentation*: Given an image, group its pixels into clusters based on their color.
4. *Astronomical data analysis*: Given a set of stars, group them into clusters based on their brightness.

6.1 k-means Clustering

A simple algorithm for clustering into k clusters. The intuition is the following: every datapoint is close to the centre of the mean of its assigned cluster. This means that once we fix the *cluster centres*, it is very easy to complete the cluster (assign each point to the cluster whose centre is closest to it).

Here is the key point: a good heuristic for the cluster centres is to take the mean of the points in the cluster. Indeed, the mean minimizes the sum of squared distances to all the points in the cluster (does that hint at what a good loss could be?).

So what we could do is: pickup random points as the cluster centres, and repeat the following two steps until convergence:

1. Assign each point to the cluster whose centre is closest to it.
2. Update each cluster centre to be the mean of the points in the cluster.

This alternates between two steps: *assignment* (computing the cluster corresponding to a set of clusters) and *update* (update the cluster centres based on the cluster). This is called the *K-means algorithm*. In a way, it is similar to Generalized Policy Iteration (GPI) in Reinforcement Learning - where we alternate between *policy evaluation* (computing the values corresponding to a policy) and *policy improvement* (update the policy based on the values). Indeed, all these algorithms are instances of *Expectation Maximization* (EM).

Let us make this precise. Let $X = \{x_1, \ldots, x_n\}$ be a set of points in $\mathbb{R}^d$, a clustering $C_1, \ldots, C_k$ and a set of cluster centres $\mu_1, \ldots, \mu_k$. Define $C^i_j := \mathbb{1}_{[x_i \in C_j]}$.

Goal: find $\mu_1, \ldots, \mu_k$ such that the sum of squared distances of each point to its clustser is minimized. That is, we want to solve the following optimization problem:

$$\min_{\mu_1,\ldots,\mu_k} \min_{C_1,\ldots,C_k} \sum_{i\in[n]} \sum_{j\in[k]} C^i_j \|x_i - \mu_j\|^2 \tag{6.1}$$

The expression is precisely our loss function $\mathcal{L}$.

$$\mathcal{L} = \sum_{i\in[n]} \sum_{j\in[k]} C^i_j \|x_i - \mu_j\|^2 = \sum_{j\in[k]} \sum_{x\in C_j} \|x - \mu_k\|^2$$

Suppose that we fix the clustering $C_1, \ldots, C_k$. Then the optimal $\mu_1, \ldots, \mu_k$ are given by setting the derivative $\frac{\partial \mathcal{L}}{\partial \mu_j} = 0$ for all $j \in [k]$. We have

$$0 = \frac{\partial \mathcal{L}}{\partial \mu_j} = 2 \sum_{x\in C_j} (x - \mu_k) \implies \mu_j = \frac{1}{|C_j|} \sum_{x\in C_j} x$$

that is, given a cluster, the optimal cluster centre is the mean of the points in the cluster - precisely the update step in the K-means algorithm.

Now suppose that we fix the cluster centres $\mu_1, \ldots, \mu_k$. Then the optimal $C_1, \ldots, C_k$ are given by assigning each point to the cluster whose centre is closest to it. (We minimize the loss for fixed μ_i over the binary matrix C^i_j that has exactly one 1 in each row and each column.)

Clearly, the minimum is achieved at

$$C^i_j = \mathbb{1}_{\left[j = \arg\min_{j\in[k]} \|x_i - \mu_j\|^2\right]}$$

Thus, given the cluster centres, the optimal clustering is the one that assigns each point to the closest cluster centre - precisely the assignment step in the k-means algorithm.

This means that each step (assignment/update) that makes a change to at least one μ_i / C^i_j is guaranteed to strictly decrease the loss. This means that the algorithm is guaranteed to converge to a local minimum.

k-means is doing gradient descent on the loss function $\mathcal{L}$!

(However, it is not guaranteed to converge to the global minimum. In fact, it is NP-hard (need to check a lot of the k^n possible clusterings) to find the global minimum of the loss.)