Dynamic Programming and Reinforcement Learning

Week 4b: Introduction to Neural Networks

Rainer Schlosser und Alexander Kastius

Enterprise Platform and Integration Concepts

06.05.21

Yet Unsolved Issues?

Still unsolved:

- State Space Complexity
  - Many Dimensions
  - Continuous Values
- Current methods require discretization and become intractable at some point

- Continuous Control
  - Action Space might consist of continuous values as well
  - Can be discretized sometimes, which prevents us from finding the actual optimal policy

”Solution”: ANNs

Abilities of modern ANNs:

- Feature extraction from high dimensional inputs
- Successful analysis of n-D input data, for example images, videos and other media types without explicit feature declaration
- Big steps towards generalization, which allows adequate performance on unknown but structurally similar inputs if used correctly
- Flexibility with regard to both their inputs and outputs

Disadvantages?

- High number of parameters requires specialized training algorithms
- Many hyper-parameters which highly influence learning performance
- Black-box behavior, unsuitable whenever predictable requirement is a must-have
- Oh, and they are hungry for data to an unprecedented level

Regression

Basic setup of a normal regression task:

- **Input definition**, consisting of the specification of the structure of an input vector or scalar x and an output vector or scalar y
- **A set of training data**, which consists of examples for both x and y, which form sets X and Y
- **A model**, that incorporates some parameters ϕ and that maps an example of x to a prediction of what the corresponding y is. Simplest example: Linear mapping, which assumes ϕ to contain a weights vector ϕ_w and a scalar parameter ϕ_b

$$y' = f(x; \phi) = x\phi_w + \phi_b$$

- A measure for the **error of the estimation, called loss function,** the determines "how wrong" the computed y' is in comparison to the actual y given a specified value of ϕ
- **An optimization algorithm**, which allows the adjustment of ϕ to minimize the measured error

Linear Models

Basic setup of a normal regression task:

- **Input definition**: A vector x and a scalar output y
- **A set of training data**: $X = (x^{(1)}, \dots, x^{(n)})$ and $Y = \left(y^{(1)}, \dots, y^{(n)}\right)$, both containing n elements
- **The model:** $y' = f(x; \phi) = f(x; \phi_w, \phi_b) = x\phi_w + \phi_b$
- **The loss function:** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2$ (This is the mean squared error, there are other alternatives around)
- **An optimization algorithm?!** (We'll check out that one later)

The Need for Non-Linear Models

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}, Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, n = 4$$

The Need for Non-Linear Models

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}, Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}, n = 4$$

There is **no** value of ϕ in a linear model

$$f(x; \phi) = x\phi_w + \phi_b$$

that can represent this.

But there is in it's non-linear two-layered big brother:

$$f(x; \phi) = \phi_{w_2} relu\left(x\phi_{w_1} + \phi_{b_1}\right) + \phi_{b_2}$$

$$relu(x) = \max(0, x)\ applied\ element\ wise$$

$$\phi_{w_1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \phi_{b_1} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \phi_{w_2} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \phi_{b_2} = 0$$

The Need for Non-Linear Models

But there is in it's non-linear two-layered big brother:

$$f(x;\phi) = \phi_{w_2} relu\left(x\phi_{w_1} + \phi_{b_1}\right) + \phi_{b_2}$$

$$relu(x) = \max(0, x)\, el.wise$$

$$\phi_{w_1} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \phi_{b_1} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \phi_{w_2} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \phi_{b_2} = 0$$

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} Y^{after-w1} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} Y^{pre-relu} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} Y^{after-relu=} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} Y^{after-w2} = Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

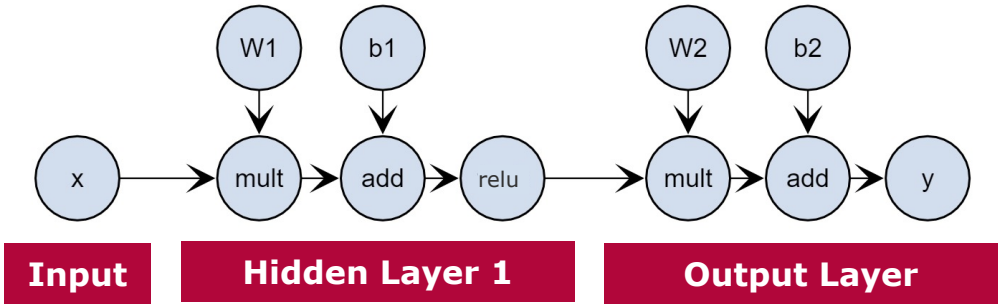
Conclusion: We can compute more with non-linear functions.

Everything, to be exact. Assuming we choose the size of the matrizes large enough.

The First Feedforward Neural Network

$$f(x;\phi) = \phi_{w_2} relu\left(x\phi_{w_1} + \phi_{b_1}\right) + \phi_{b_2}$$
$$relu(x) = \max(0, x)\ el.wise$$

Representation as a graph:



From here on, we can: Exchange the **non-linear function**, adjust the **format of the matrices** (and thus adjust the size of the output of each layer) or **add more layers**!

Recall: Linear Models

Basic setup of a normal regression task:

- **Input definition**: A vector x and a scalar output y
- **A set of training data**: $X = (x^{(1)}, \dots, x^{(n)})$ and $Y = \left(y^{(1)}, \dots, y^{(n)}\right)$, both containing n vectors
- **The model:** $y' = f(x; \phi) = f(x; \phi_w, \phi_b) = x\phi_w + \phi_b$
- **The loss function:** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2$ (This is the mean squared error, there are **other alternatives** around)
- **An optimization algorithm?!**

Loss Functions

There is more than one possible choice for the loss function:

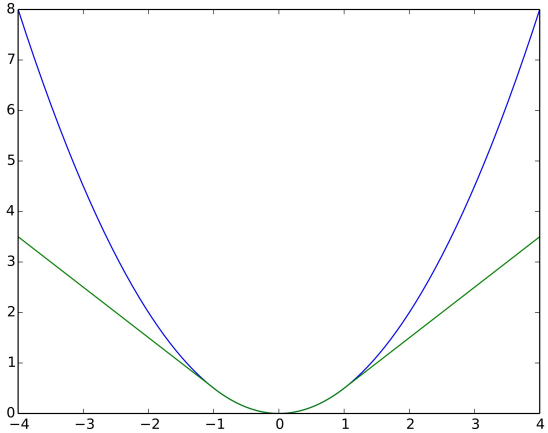
- **Mean Squared Error:** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2$
- **Huber Loss:** $L(X, Y; \phi) =$

$$\frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2 \; for \left|f\left(x^{(i)}; \phi\right) - y^{(i)}\right| < \delta, \\ \delta\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right) - \frac{1}{2}\delta^2 \; otherwise\end{cases}$$

  (Less influenced by outliers than MSE)

- **Cross-Entropy** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(-\sum_{j=1}^{h} y_j^{(i)} \log\left(f\left(x^{(i)}; \phi\right)_j\right)\right)$

  (Well suited if the output of the network represents a probability distribution (the output is a vector over all possible choices of length h) and $y^{(i)}$ contains the ground truth)



Huber (blue) vs MSE (green),

Src.: https://en.wikipedia.org/wiki/Huber_loss

Recall: Linear Models

Basic setup of a normal regression task:

- **Input definition**: A vector x and a scalar output y
- **A set of training data**: $X = (x^{(1)}, \dots, x^{(n)})$ and $Y = \left(y^{(1)}, \dots, y^{(n)}\right)$, both containing n vectors
- **The model:** $y' = f(x; \phi) = f(x; \phi_w, \phi_b) = x\phi_w + \phi_b$
- **The loss function:** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2$
- **An optimization algorithm?!**

Simply Solving ANNs

- To solve the system of equations given by the parameters for the correct weights is usually intractable.
- This is caused by the most prominent disadvantage of ANNs: Very large search spaces.
- Every layer with p inputs and o outputs introduces $p \times (o + 1)$ parameters.
- In many examples, this is a huge number. Working on images with 256 by 256 pixels causes an input size of 65536 dimensions in the first layer.
- Output widths of 256 or larger in hidden layers are not unusual for networks charged with problems in high-dimensional input spaces.
- Output layers of Deep Q-Networks can have as many outputs as there are actions in the process which is observed.
- **We need a different solution.**

Gradients on a Linear Model

- We have an example of ϕ, generated randomly within the allowed values.
- We can now compute the loss function by iterating through the whole training set once.
- More importantly, we can compute:

$$\nabla_{\phi} L(X, Y; \phi)$$

- ∇_{ϕ} is the gradient over all elements in ϕ, consisting of all partial derivatives with regard to the respective parameter
- This points us in the direction we could adjust every single one of them to minimize (descent) or maximize (ascent) the error by changing ϕ
- Machine learning libraries can compute this value for f of arbitrary structure.
- This requires f and L to be differentiable everywhere (which $relu$ isn't, but we can do an approximation of it that is differentiable everywhere)

Gradients on a Non-Linear Layered Model aka. Backpropagation

- To compute the output of our ANN given x we iterate through all matrices forward, a process called forward-propagation.
- To compute the gradient with regard to a certain weight or a bias (which is a weight with regard to a 1-input), we need to reverse this operation.
- The computation of the **gradient of the loss** with regard to ϕ becomes straightforward if we know the **gradient of the output** of the network with regard to ϕ.
- Assuming the activation function of the network is linear in the last layer, the gradient with regard to the weights of the last layer ($l_j(x_j; \phi_{w_j})$, x_j is the output of l_{j-1}, $j = 1, \dots, num\ of\ layers$) becomes:

$$\nabla_{\phi_j} l_j(x_j; \phi_j) = \nabla_{\phi_j} x_j w_j$$

- Assuming we know the derivatives with regard to the output layer, the weights of the inner layers behave similarly, with the limitation that knowledge about the gradients of the following layer is required. The **chain rule** then allows the computation of the gradients on weights in hidden layers.

Gradient Descent

- It becomes possible to compute the value of the gradient for all elements of ϕ through back-propagation.
- We can then use this gradients to improve the performance of our network by computing the gradient and adding its negative value to the parameters (we want to minimize the loss, not maximize it).

1. We initialize ϕ_0 with random values. ϕ_k denotes ϕ after iteration $k, k \geq 0$
2. We compute the loss function and the gradient ∇_ϕ for the whole set of training data.
3. We update the parameters according to, given learning rate $\eta > 0$:

$$\phi_{k+1} = -\eta \nabla_{\phi_k} L(X, Y; \phi_k) + \phi_k$$

Minibatch Stochastic Gradient Descent

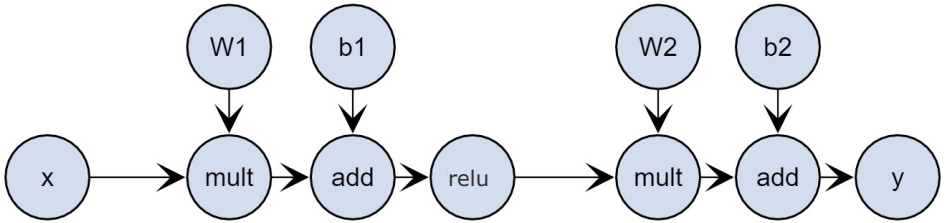
- The previous method is tedious, as it requires computation of the gradients according to the whole training set.
- We can take advantage of a small adjustment: The expected value of the gradient of a sampled example of elements taken from the training set is equal to the actual gradient of the whole training set.
- Due to this, we can sample a small set and update the parameters earlier and repeat the process more often.

1. We initialize ϕ_0 with random values.
2. We sample a minibatch of data from the training data.
3. We compute the loss function and the gradient ∇_ϕ for the batch.
4. We update the parameters according to $\phi_{k+1} = -\eta \nabla_{\phi_k} L(X, Y; \phi_k) + \phi_k$
5. Repeat from 2.

Final Setup: ANNs

Basic setup of a normal feedforward neural network:

- **Input definition**: A vector x and a scalar output y
- **A set of training data**: $X = (x^{(1)}, \ldots, x^{(n)})$ and $Y = \left(y^{(1)}, \ldots, y^{(n)}\right)$, both containing n elements
- **The model:**



- **The loss function:** $L(X, Y; \phi) = \frac{1}{n}\sum_{i=1}^{n}\left(f\left(x^{(i)}; \phi\right) - y^{(i)}\right)^2$
- **The optimization algorithm:** Stochastic Gradient Descent (SGD)

Recall: ANNs Advantages and Disadvantages

Abilities of modern ANNs:

- Feature extraction from high dimensional inputs
- Successful analysis of n-D input data, for example images, videos and other media types without explicit feature declaration
- Big steps towards generalization, which allows adequate performance on unknown but structurally similar inputs if used correctly
- Flexibility with regard to both their inputs and outputs

Disadvantages?

- High number of parameters requires specialized training algorithms
- Many hyper-parameters which highly influence learning performance
- Black-box behavior, unsuitable whenever predictable requirement is a must-have
- Oh, and they are hungry for data to an unprecedented level

Extensions Possibly Relevant for the Project

Convolutional Networks

Apply a folding mechanism to the n-dimensional input data, which repeatedly searches for patterns.

Very beneficial when applied to problems with screenshots or image output. This occurs regularly in robotics or many experimental environments for reinforcement learning.

LSTMs

Long Short-Term Memory Networks receive their own output as input, which gives them the ability to "remember" what they've previously seen.

Perform well when challenged with a task where previously seen elements do matter. This might be the case in processes which hide information in the state which was easily observable in previous states.

Better optimizers

Algorithms like **ADAM** do take advantage of some properties of mini-batch SGD, which allows much faster convergence if the gradients repeatedly point in the same direction.

Do work well in basically any scenario, should be applied whenever possible.

Application of ANNs with TensorFlow/Keras

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10),
])

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10)

model.evaluate(x_test,  y_test, verbose=2)
```

Schedule

- 3.5.: Problems with Q-Learning
- **10.5.: Deep Q-Networks**
- 17.5.: DQN Extensions (Rainbow)
- 27.5.: Policy Gradients (2)
- 31.5.: Project Assignments
- 7.6.: Project Work/Support
- 14.6.: Project Work/Support
- 21.6.: Project Work/Support
- 28.6.: Project Work/Support
- 5.7.: Project Work/Support
- 12.7.: Final Presentations
- 31.8.: Final Documentation

6.5.: Artificial Neural Networks

20.5.: Policy Gradients (1)

3.6.: Project Assignments

10.6.: Project Work/Support

17.6.: Project Work/Support

24.6.: Project Work/Support

31.6.: Project Work/Support

8.7.: Project Work/Support

15.7.: Final Presentations