MATERIAL FOR NEURAL NETWORKS
MACHINE LEARNING E20

Mathias Ravn Tversted
Department of Computer Science
Aarhus University
Åbogade 34, 8200 Aarhus N
Tversted@post.au.dk

December 27, 2020

Contents

1 Introduction

Deep Learning uses neural networks to make simple and complex feature extraction from images and maps these features to different outputs.
One of the most common uses is image classification, but there are a lot of different genres all from image generation to image transformation.

2 Neural Network

A Neural Network is a connection of nodes that goes from some inputs and some biases to some outputs.
These connections have different weights which is what we try to make better dudring training.
To add more complexity and control each node can execute an activation function.

3 Training neural networks

IKKE SIKKERT AT DET SKAL VÆRE LIGE PRÆCIS HER

Training neural networks is usually done with techniques such as *(stochastic/mini-batch) gradient descent*, with any point-wise loss function $E_{in}(w) = \frac{1}{n}\sum_{i=1}^{n} L(h(x_i), y_i)$

Mini-batch SGD:

- Initialise w (e. g random $N(0, 1)$)
- While more time (epochs may be fixed):
- 1. Shuffle $(x_1, y_1), \cdots, (x_n, y_n)$
  2. For $i = 1, \cdots, n/B$:
  3. Let $(x'_1, y'_1), \cdots, (x'_n, y'_n)$ be the next batch
  4. $g = \frac{1}{B}\sum_{i=1}^{B} \nabla L(w(x'_i), y'_i)$
  5. $w = w - \eta g$

4 Activation functions

Activation functions are very simple.
Common ones are:

- ReLU (Rectified Linear Unit): $f(x) = max(0, x)$
- Sigmoid: $\sigma(z) = \frac{1}{1+e^z}$

Sigmoid squishes outputs into the intervl $[0, 1]$, which is useful for networks that require probabilities as outputs.

ReLU is useful in that it allows neurons to be "turned off" if they do not receive sufficient output. This is useful, as it allows for flexibility in training, and for certain neurons to do certain things, rather than always having to contribute to the output.

5 Forward/Backward Pass

When we evaluate a Neural Network we call it a Forward Pass. That simply follows the activation rules given in each node and passes the values added together forward.
These Forward Passes are differentiable so we can calculate which way we should move the different weights in order

to get better reuslts. This process is called a Backward Pass.

HER KOMMER ET EKSEMPEL A simple neural network from TA Class: $nn(x_1) = w_3 * ReLU(w_1 * x_1 + w_2 * x_1 + 3)$.
We wish to optimize this using least squares error: $e = (y - nn(x))^2$
First we do the forward pass for some data (x, y) and some initial weights. (The values at the end of the line after # are the calculated values.)

```
x1 = 3.0
y = 9.0
w1 = 1.0
w2 = 2.0
w3 = 1.0
# Long Forward Pass
z1 = w1 * x1 #3.0
z2 = w2 * x1 #6.0
hin = z1 + z2 + 3 #12.0
hout = max(hin, 0) #12.0
pred = w3 * hout #12.0
diff = pred - y #3.0
e = diff**2 #9.0
```

And then we do the Backward Pass which uses many of the values calculated in the Forward Pass.

```
de_diff = 2*diff #6.0
ddiff_pred = 1 #1.0
de_pred = de_diff*ddiff_pred #6.0
dpred_w3 = hout #12.0
de_w3 = de_pred * dpred_w3 #72.0
dpred_hout = w3 #1.0
de_hout = de_pred * dpred_hout #6.0
dhout_hin = 1 if hin > 0 else 0 #1.0
de_hin = de_hout * dhout_hin #6.0
dhin_z1 = 1 #1.0
de_z1 = de_hin * dhin_z1 #6.0
dhin_z2 = 1 #1.0
de_z2 = de_hin * dhin_z2 #6.0
dz1_w1 = x1 #3.0
de_w1 = de_z1 * dz1_w1 #18.0
dz2_w2 = x1 #3.0
de_w2 = de_z2 * dz2_w2 #18.0
```

6 Fully connected networks

We can have what is called fully connected networks. This means that all nodes in one layer are connected to alle nodes in the next layer.
This can make computation very effective since we can multiple big vector and matrices together all at once using GPU acceleration.
Our computation then has the following form where ϕ are the activation functions for a layer, x is the input, W are the weights for the connections between two layers, and b are the biasis for a layer:
$\phi^k(\phi^{k-1}(\dots\phi^2(\phi^1(xW^1 + b^1)W^2 + b^2)\dots)W^k + b^k)$

7 Convolutions

Convolutions act as a kind of "filter" on images that can be *trained* to identify (more abstract) features. A $k \times k$ convolution is specified by a $k \times k \times c$ tensor, where k is often odd, so it can center on a pixel. The idea is to "slide" it across the image with some stride s. (Tells us how much we shift).

An entry in the output tensor is then a linear combination of the entries in the corresponding convolution window as well as the weights of the convolution. The weights are shared.

Example: Edge detection:

$$E = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1}$$

Which checks if the middle pixel is different from average. This is *translation invariant* as it finds the same pattern. This is important, because these kinds of convolutions allow us to find the same things in different images.

Convolutions reduce the size of images, but with $\frac{k-1}{2}$ 0s as padding, it can center on every pixel and produce the same output size.

Scale-invariance however, could be a problem.

7.1 Pooling

Is a $k \times k$ function of k elements $\phi : R^k \to R$ with stride s. For example, max-pooling where the output is the maximum value in the window. There is also mean and min. Pooling with stride $s > 1$ reduces image size by factor s.

8 overfitting

With enough states a neural network can learn to produce all the desired outcomes, but might not generalize that well to new data. There are multiple methods that can help mitigate this overfitting of the training data.

8.1 Validation - Early Stopping

You can use a validation set to test the model while training it. Then when you see a increase in the validation that means that the model is beginning to overfit. You can stop there or in practice check for a multiple epochs and if it keeps increasing then stop and use the model that had the best validaiton error. Normally the validation set is about 20% of the test set.

8.2 Regularization - Weight Decay

Sometimes the weights tend to skyrocket to fit the training set better. We can fix this by punishing big weights as a part of the in-sample error function.
$E_{in}(nn) = \frac{1}{n}\sum_{i=1}^{n} L(nn(x_i), i) + \lambda \sum_{w \in nn} w^2$. We use the variable λ to specify how much it should punish big weights. A good value is around 0.01 we found from tests in the second handin.

8.3 Dropout - Not pensum

We can modify our Neural Network so it has some chance of not using the value from a node. This is called dropout and is very effective, but not very technical to implement.

9 Cross-entropy loss

Recall that for multinomial regression, we use softmax as our loss function

$$L(h(x), y) = -ln(y^T softmax(x^T W)) \tag{2}$$

With K entries summing to 1. For multiclass classification however, this can be problematic, as the target will contain 1 in several entries.

If we want to do multi-class classification, we can use cross-entropy loss as our loss function. This can be considered a generalisation of softmax for multinomial regression.

Entropy is defined as $-H(S) = \sum_{i=1}^{k} p_i log_2(p_i)$, but if we substitute $log_2(p_i)$ for our hypothesis $h(x)$, we will gain higher values as our hypothesis diverges from the actual probabilities. Cross entropy loss is then defined as $-L(h(x), y) = \sum_{j=1}^{K} y_i ln(h(x_i)_j)$

The In-sample Error is defined using Cross-entropy loss for K-label neural networks as follows:
$E_{in}(h) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{K} y_{i,j} ln(h(x_i)_j)$

It also means that we can remove softmax layers and include it in the loss function instead.