Advanced course in machine learning
582744
Lecture 10

Arto Klami

Neural networks

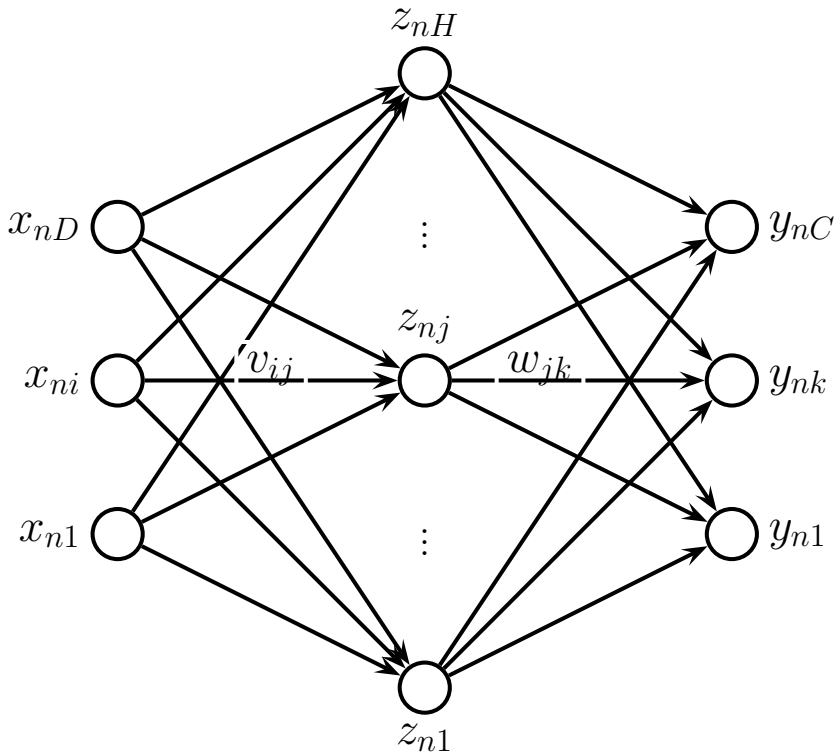
Class of methods loosely motivated by how the brain might work. A more practical definition is that they are non-linear models built from large number of simple modules

We have already seen some neural networks: Self-organizing maps and Radial basis function models

We will spend most of the time discussing the canonical neural nwtwork model, *multilayer perceptron* (MLP), also called feedforward neural network

Neural networks are specific models for unsupervised or supervised learning, here treated separately in part because of the current *deep learning* hype

Multiplayer perceptrons



Each *unit* (or neuron) takes as input a weighted sum of the outputs of the previous layer and passes it through some *activation function*

The layers between the inputs and outputs are *hidden layers* (or latent variables as we would call them outside MLPs)

Activation functions

- Heaviside step-function: $|u|$
- Sigmoid or tanh: $s(u) = (1 + e^{-u})^{-1}$, $\tanh(u) = 2s(2u) - 1$
- Rectified linear (ReLU): $\max(0, u)$

The first used as historical motivation for how actual neurons might work – they "fire" when the input is large enough

Sigmoids are simply convex and smooth approximations for that, and the rectified linear unit is what people often use nowadays since it does not require second-order information for fast learning

Why non-linearity?

Denote the activation function for the hidden level by $g(u)$ and for the output player by $h(u)$

One output of the example MLP is $h(\mathbf{w}_k^T \mathbf{z}) = h(\mathbf{w}_k^T g(\mathbf{V}\mathbf{x}))$, which is actually an adaptive basis function model – each neuron at the hidden level is one basis function

If $h(u) = g(u) = I(u)$ then the above becomes $\mathbf{w}_k^T \mathbf{V}\mathbf{x}$, which is equivalent to $\nu^T \mathbf{x}$ for $\nu = \mathbf{V}^T \mathbf{w}_k$ – the output is linear and hence nothing was gained by adding the hidden layer

Flexibility

Universal approximation theory says MLPs can approximate any suitably smooth function with arbitrary precision, assuming non-linear activiation functions

This holds already for MLPs with single hidden layer

Sketch of the proof for univariate inputs and outputs: Use plenty of hidden nodes with slightly different biases to create pairs of hidden nodes that carve out a small piece of the input

Pretty much all adaptive basis function models satisfy this...

Why several layers?

If already one hidden layer is enough, why bother using more?

It is often considerably easier to learn the function with more layers – we can think of the layers as progressive feature extractors (and can even train the networks layer by layer)

Might require less parameters in total and hence reduce overfitting

`http://playground.tensorflow.org`

The output layer

The output layer is simply some generalized linear model, and hence linear regression, logistic regression, etc are MLPs with no hidden layers

We can think of MLPs as adaptive basis function models where all of the earlier layers generate the basis

The activation function of the output is determined by the task, and hence typically is different than the activation function for the hidden layers

To implement multiclass classification, we need mutual inhibition arcs between the output nodes

Probabilistic formulation

Most losses correspond to some probability density over the labels, and hence probabilistic interpretation of MLP is clear

Learning by maximum likelihood or by maximum a posterior if specifying priors for the weights

Hence: Neural networks (and deep learning in general) are simply specific family of probabilistic models, with considerable attention dedicated to finding the ML estimate of highly non-convex loss

Learning MLPs

Learning a MLP requires both fixing its structure (the number of the layers, the activation functions, the number of nodes in the layers, ...) and learning the weights

We consider first the latter, and return to the (harder) problem of determining the structure afterwards

The weights we naturally learnt by minimizing some loss function using gradients

Backpropagation algorithm

The challenge is that we only have targets for the last layer, so we need to propagate the gradients back through the network

In the end this is merely an exercise in the chain rule of derivation, yet it took until mid-eighties for people to formulate it despite the model dating back to the sixties

Consider a case with just one hidden layer:

$$\mathbf{a} = \mathbf{V}\mathbf{x}, \qquad \mathbf{z} = g(\mathbf{a}),$$
$$\mathbf{b} = \mathbf{W}\mathbf{z}, \qquad \mathbf{y} = h(\mathbf{b})$$

i indexes inputs, j indexes hidden units and k indexes outputs; V_{ji} links $\mathbf{x}_i$ to $\mathbf{z}_j$; $\mathbf{W}_{kj}$ links $\mathbf{z}_j$ to $\mathbf{y}_k$

Backpropagation

Consider first the gradient wrt to the output layer weights

$$\frac{\partial L}{\partial \mathbf{w}_{kj}} = \frac{\partial L}{\partial \mathbf{b}_k} \frac{\partial \mathbf{b}_k}{\partial \mathbf{w}_{kj}} = \frac{\partial L}{\partial b_k} \mathbf{z}_j$$

The first term is

$$\frac{\partial L}{\partial \mathbf{b}_k} = \frac{\partial L(\hat{\mathbf{y}}_k, h(\mathbf{b}_k))}{\partial h(\mathbf{b}_k)} \frac{\partial h(\mathbf{b}_k)}{\partial \mathbf{b}_k}$$

For squared loss we use $h(\mathbf{b}_k) = I(\mathbf{b}_k)$ and get the derivative $\hat{\mathbf{y}}_k - h(\mathbf{b}_k) = \hat{\mathbf{y}}_k - \mathbf{y}_k$

Backpropagation

Binary classification:

- $L = -\hat{y} \log y - (1 - \hat{y}) \log(1 - y)$
- Sigmoid activation $h(b) = (1 + e^{-b})^{-1}$
- Derivative $\hat{y} - h(b) = \hat{y} - y$ (Exercise 2)

For a wide range of losses we can find the canonical link function that results in $\frac{\partial L}{\partial \mathbf{b}_k} = \hat{y}_k - y_k \equiv \boldsymbol{\delta}_k$

This is largely what generalized linear models (the concept we skipped) are about, but here it is enough to know it holds for the above cases

The whole gradient wrt to the output layer weights is hence simply the error weighted by the activations of the hidden layer

$$\nabla_{\mathbf{w}} L = \boldsymbol{\delta}_k \mathbf{z},$$

Backpropagation

How about the weights for the hidden layer?

$$\frac{\partial L}{\partial \mathbf{V}_{ji}} = \frac{\partial L}{\partial \mathbf{a}_j}\frac{\partial \mathbf{a}_j}{\partial \mathbf{V}_{ji}} = \frac{\partial L}{\partial \mathbf{a}_j}\mathbf{x}_i$$

To compute the first term we need to sum over the outputs that depend on $\mathbf{a}_j$

$$\frac{\partial L}{\partial \mathbf{a}_j} = \sum_k \frac{\partial L}{\partial \mathbf{b}_k}\frac{\partial \mathbf{b}_k}{\partial \mathbf{a}_j} = \sum_k \boldsymbol{\delta}_k \frac{\partial \mathbf{b}_k}{\partial \mathbf{a}_j}$$

Since $\mathbf{b}_k = \sum_j \mathbf{W}_{kj} g(\mathbf{a}_j)$, the latter term is

$$\frac{\partial \mathbf{b}_k}{\partial \mathbf{a}_j} = \mathbf{W}_{kj} g'(\mathbf{a}_j)$$

Backpropagation

In the end we hence get

$$\frac{\partial L}{\partial \mathbf{V}_{ji}} = \left(\sum_k \boldsymbol{\delta}_k \mathbf{W}_{kj} g'(\mathbf{a}_j) \right) \mathbf{x}_i$$

Compare this to the output layer

$$\frac{\partial L}{\partial \mathbf{W}_{kj}} = \boldsymbol{\delta}_k \mathbf{z}_j$$

Denote

$$\boldsymbol{\delta}_j^{(z)} = \left(\sum_k \boldsymbol{\delta}_k \mathbf{W}_{kj} g'(\mathbf{a}_j) \right)$$

to see they are equal – the latter just uses an error signal that was propagated back using the weights $\mathbf{W}$ multiplied with the gradient of the activation function

What if we have more layers?

For a MLP with two hidden layers the derivatives wrt to the input weights go through two nested summations, since we need to cover all routes to the outputs

The key idea of backpropagation is that we do not need to do this explicitly. Instead, we can just propagate the error signal back and then act if the latter layers did not exist.

In other words, the equations above generalize for arbitrarily deep networks by induction

Backpropagation

Practical computation combines forward and backward passes:

- Forward pass: Each unit multiplies the inputs at the previous layer by the weights and passes the value through the activation function
- Backward pass: Each unit multiplies the errors coming from the next layer by the weights and further multplies by the gradient of the activation function

Stochastic gradients easy – the derivation above was anyway for a single sample. Second-order techniques also possible, but more cumbersome

In practice we use *automatic differentiation* to compute the gradients; they anyway consists of sums of gradients of elementary functions

Initialization of MLPs

Random weights are okay, but need to be small enough so that the units initially operate roughly in the linear region of the activation functions

Can also scale wrt to the number of incoming and outcoming links, so that the variance of both the forward and backward passes is roughly retained at each layer

Remember from Exercise 2 that for the logistic function the gradient almost vanishes when the input is very large (or very small) – we want to avoid that at least in the beginning

Regularization of MLPs

MLPs are flexible models and hence prone to overfitting

Regularization by

- Early stopping – if we initialize with small weights the model is linear in the beginning and becomes more complex during the iteration
- Weight decay – l_2 regularization on the weights; specific name for historical reasons
- Weight sharing – reduce number of parameters by forcing some nodes to use the same weights, or by encouraging them to be similar by shared prior
- Weight pruning – replace small weights with zeroes to approximate l_0 regularization, or just use l_1 prior

We will later discuss specific modern regularization techniques

History of neural networks

- Mathematical model for neurons by McCulloch&Pitts (1943)
- Perceptron algorithm by Rosenblatt (1957)
- Minsky&Papert (1969): Perceptrons (with no hidden layers) only solve linearly separable cases
- Rumelhart&Hinton&Williams (1986) invented backpropagation
- LeCun et al. (1989): LeNet – a practical MLP that solved an interesting problem (we will come back to this next lecture
- SVMs (1992) were as accurate but with convex loss functions, stealing the stage
- Layer-wise training (2002) slowly restarted the interest in neural networks, with strong hype since 2010 or so – this is largely because we can now solve harder optimization problems