Machine Learning in Finance
Section I3
Lecture 11

Michael G Sotiropoulos

NYU Tandon
Deutsche Bank

19-Apr-2019

Contents

Overfitting Control: Regularization (I)

ANNs introduce a lot of tunable parameters (weights and biases).
Consider a dense ANN with M input nodes, K output nodes, and D hidden layers with nodes $H_1, H_2, \dots H_D$. The total number of parameters becomes

$$\underbrace{M \cdot H_1 + H_1 \cdot H_2 + \dots H_{D-1} \cdot H_D + H_D \cdot K}_{\text{weights}} + \underbrace{(H_1 + H_2 + \dots H_D)}_{\text{biases}} \tag{1}$$

This can easily reach thousands or millions. ANNs are vulnerable to overfitting.

Regularization is used to constrain the size of the weight vectors only. Biases are typically not regularized, since this may lead to underfitting.
The modified loss function contains a shrinkage (penalty) term

$$\tilde{L}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = L(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) + \alpha \Omega(\boldsymbol{w}) \tag{2}$$

Similarly to earlier ML algorithms we can use L^1 or L^2 regularization.

$$\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i| \tag{3}$$

$$\Omega(\boldsymbol{w}) = \|\boldsymbol{w}\|_2 = \left(\sum_i w_i^2\right)^{1/2} \tag{4}$$

Overfitting Control: Regularization (II)

Consider a specific hidden layer d, with H_d nodes (units). Its weights can be represented by kernel (matrix) W_{ij}, where $i = 1, 2, \ldots H_{d-1}$ and $j = 1, 2, \ldots H_d$. L^2 regularization on this layer would use the shrinkage term $\alpha\Omega(\boldsymbol{W})$ with

$$\Omega(\boldsymbol{W}) = \left(\sum_{ij} W_{ij}^2\right)^{1/2} = \sqrt{\mathrm{trace}\left(\boldsymbol{W}^T \cdot \boldsymbol{W}\right)} \tag{5}$$

This is called the **Frobenius norm** of the matrix.

- In ANN practice an alternative normalization is used, in which the norm of **each column** of $\boldsymbol{W}$ is constrained separately.
- This is to prevent any individual node (column) from dominating the rest.

Keras supports L^1, L^2 and combination $L^1 - L^2$ regularization (like elastic net) in `keras.regularizers`. Each layer can be regularized individually. Dense and activated weights can be regularized separately.

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

Overfitting Control: Dropout (I)

Dropout is the most widely used ANN regularization technique. Despite its simplicity it works quite well in practice.

- Dropout is an efficient way of approximating bagging, using ANN ensembles with exponentially increasing number of networks.

It works as follows:

1. Divide the training set into mini-batches. Each mini-batch is processed as a training step.
2. At each training step a non-output node (input or hidden) is omitted with probability p
3. Nodes to be omitted are sampled independently from other nodes

Dropout can be implemented by creating a mask vector $\boldsymbol{\mu}$ with as many components as non-output nodes. Each component of $\boldsymbol{\mu}$ can be either 0 or 1. At each training step the vector $\boldsymbol{\mu}$ is sampled, and each node i is multiplied by μ_i, effectively switching the node to on/off.

- If we use a distribution $p(\boldsymbol{\mu})$ to sample the mask vector, the ANN estimates a posterior

$$p(\boldsymbol{y}|\boldsymbol{X}) = \sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu})\, p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\mu}) \tag{6}$$

Overfitting Control: Dropout (II)

Below we consider a network with two input and two hidden units.
All possible dropout combinations ($2^4 = 16$) create the ensemble subnets.
The original network is trained with each node being multiplied by the mask vector component μ_i. There are 16 possible binary vectors $\boldsymbol{\mu}$.

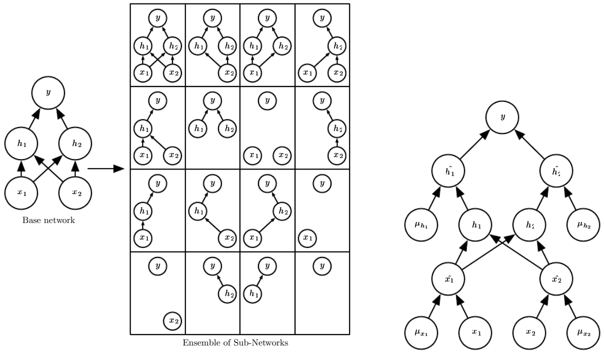

Figure: Original ANN (left), all possible dropout subnets (middle), and masked ANN (right). Figures taken from [GBC].

Overfitting Control: Dropout (III)

In practice, the probability p is called the **dropout rate**.
Common values are 0.8 for input nodes and 0.5 for hidden nodes.
Dropout occurs during the training phase **only**, not during testing/prediction.

Important technical point: if during training the dropout rate was p, then during testing each input connection needs to be multiplied by the keep probability $1 - p$. Otherwise, a node will be receiving $1/(1 - p)$ more input during testing relative to what it was trained with.

In Keras, dropout is implemented as a core layer, in class `keras.layers.Dropout`

- The layer is added *after* an input or hidden layer.
- The constructor accepts the `rate` parameter, between 0 and 1.
- The parameter `noise_shape` controls the shape of the mask.
  This allows applying different masks to different dimensions of the input, useful for image data.

Overfitting Control: Early Stopping

Early stopping is applied during the training phase. It requires a validation set.

Initially, as the ANN iterates over the training set, fitting error decreases both on the training and validation sets. But with more epochs, training error keeps decreasing, while validation error stays flat or starts increasing. This is clear symptom of overfitting. Early stopping works as follows:

- Monitor performance on the validation set, and stop training once validation performance stops improving or deteriorates.

Early stopping is used in combination with the previous overfitting controls.

In Keras this is implemented using `keras.callbacks.EarlyStopping.`

1. Create an instance of the class as: `es=EarlyStopping(monitor=...)`
2. Pass the instance to the fit method as: `model.fit(..., callbacks=[es])`

Important constructor parameters

- `monitor`: sets the performance metric to track
- `min_delta`: defines the minimum absolute change for continuing training
- `baseline`: performance level to be exceeded otherwise training stops
- `restore_best_weights`: if true it restores the best fitting weights from the training history before exiting

Optimizers: SGD

Training an ANN means solving a high dimensional optimization problem. Let's collect weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ into a long vector of parameters $\boldsymbol{\theta}$. Given data $(\boldsymbol{x}, \boldsymbol{y})$, the loss function is $L(\boldsymbol{\theta})$. The problem becomes

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{7}$$

Using reverse autodiff, the algorithm computes the gradients $\boldsymbol{g} = \nabla_\theta L$ at the current value of $\boldsymbol{\theta}$. This is called back-propagation (**backprop**).
A gradient descent (GD) step with learning rate η does the update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g} \tag{8}$$

The stochastic part of SGD refers to the choice of updating a random subset of the components of $\boldsymbol{\theta}$.

What can go wrong with SGD?

- It may be too slow to converge, i.e the steps $\eta\boldsymbol{g}$ may be too small.
- When training deep ANNs, some of the components of $\boldsymbol{g}$ may become too small (vanishing) or too bid (exploding).
- Vanishing gradients make the optimizer get stuck in flat valleys.
- Exploding gradients make the optimizer numerically unstable.

We need to find ways to improve standard SGD.

Optimizers: Momentum (I)

Momentum optimization (Polyak 1964) uses the idea that the steps should be larger if *earlier* gradients were large.
Rolling down a steep slope should build up momentum towards the bottom.
Note that SGD has no memory of earlier gradients, the vector $\boldsymbol{g} = \nabla_\theta L$ depends only on current values.

A momentum vector $\boldsymbol{m}$ is used to maintain memory of earlier gradients. The update scheme becomes

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} + \eta \boldsymbol{g} \tag{9}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \boldsymbol{m} \tag{10}$$

The hyperparameter $0 \leq \beta \leq 1$ discounts history relative to the recent gradient (similar to EMA). It acts as a friction (break) to avoid overshooting.

In Keras, momentum (β) is a parameter used in the construction of SGD.

```
opt = keras.optimizers.SGD(lr=0.1, momentum=0.5, ...)
```

The parameter `lr` is the learning rate (η).
The optimizer `opt` is passed to the `model.compile` method.

Optimizers: Momentum (II)

An improvement over standard momentum is **Nesterov Accelerated Momentum** (NAG) or Nesterov update (Nesterov 1983).

- NAG computes the gradient not at the current point $\boldsymbol{\theta}$, but at the forward point $\boldsymbol{\theta} + \beta \boldsymbol{m}$. The update scheme becomes

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} + \eta \nabla L(\boldsymbol{\theta} + \beta \boldsymbol{m}) \tag{11}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \boldsymbol{m} \tag{12}$$

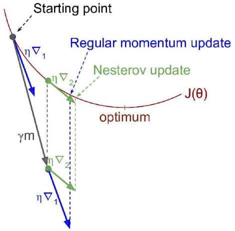

Figure: Regular and Nesterov updates. $J(\theta)$ is the loss function $L(\boldsymbol{\theta})$. Figure from [GA].

In Keras, NAG is a flag passed to the constructor of SGD.

```
opt = keras.optimizers.SGD(lr=0.1, momentum=0.5, nesterov=True)
```

Optimizers: Adaptive (I)

In standard or momentum SGD, the step direction is set by the dimensions with the largest gradients. This means we initially go quickly down steep slopes and then very slowly along shallow valleys.
Adaptive methods try to detect this *early*, and point more towards the minimum.

AdaGrad is the first attempt. It follows two steps.

1. Accumulate the squares of the components of the gradient in a vector $\boldsymbol{s}$
$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \boldsymbol{g} \otimes \boldsymbol{g} \tag{13}$$
where $\boldsymbol{g} = \boldsymbol{\nabla} \boldsymbol{L}\left(\boldsymbol{\theta}\right)$ and $\otimes$ is component-by-component multiplication
2. Scale down the gradient vector by dividing it by the scaling vector $\boldsymbol{s}$ and then update the parameters
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g} \oslash \sqrt{\boldsymbol{s} + \epsilon} \tag{14}$$
where $\oslash$ means component by component division.

This has the effect of reducing large gradient components relative to small ones. The small offset ϵ is to avoid division by zero.

- Nice idea, but in practice it may stop too early when training ANNs.

Optimizers: Adaptive (II)

RMSProp tries to fix the stalling problems of AdaGrad by accumulating only recent gradient upgrades.
Like EMA, RMSProp introduces a decay rate (discount factor) β and follows the update scheme

$$\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\, \boldsymbol{g} \otimes \boldsymbol{g} \tag{15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g} \oslash \sqrt{\boldsymbol{s} + \epsilon} \tag{16}$$

The decay rate is typically set to $\beta = 0.9$.

In Keras, RMSProp is implemented in the class `keras.optimizers.RMSProp`.

```
opt = keras.optimizers.RMSProp(lr=0.01, decay=0.9, epsilon=1.0e-10)
```

- Both AdaGrad and RMSProp are adapting (rescaling) the learning rate.
- In practice RMSProp has been found to outperform AdaGrad and Momentum for training ANNs.
- RMSProp has been the standard choice, until the arrival of the Adam class of optimizers.

Optimizers: Adaptive (III)

Adam means **Ada**ptive **M**omentum. It is a blend of Momentum and RMSProp.

Adam keeps track of both an exponentially weighted sum of past gradients like Momentum, and of past squared gradients like RMSProp.
There are two decay rates β_1 (momentum) and β_2 (RMSProp).
The Adam update scheme is

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1-\beta_1)\boldsymbol{g} \tag{17}$$

$$\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1-\beta_2)\underbrace{\boldsymbol{g} \otimes \boldsymbol{g}}_{\text{norm}} \tag{18}$$

$$\boldsymbol{m} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1} \tag{19}$$

$$\boldsymbol{s} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2} \tag{20}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{m} \oslash \sqrt{\boldsymbol{s}+\epsilon} \tag{21}$$

Eqs. (19, 20) are there to boost $\boldsymbol{m}$ and $\boldsymbol{s}$ away from their initial small values. Otherwise they are not of major importance. The other three updates are clearly a combination of momentum and RMSProp.

Optimizers: Adaptive (IV)

The main ideas in AdaGrad, and Adam have several extensions.
Here are the main variations and how they can be easily accessed as classes in the `keras.optimizers` package.

AdaGrad is implemented in the class `Adagrad(lr, epsilon)`

Adam is implemented in the class `Adam(lr, beta_1, beta_2, epsilon)`

AdaDelta is an AdaGrad variation, in which the scaling vector $\boldsymbol{s}$ is aggregated on a moving window instead of accumulating gradients from start.
The size of the effective window is determined by the decay parameter rho.
This is implemented in the class `Adadelta(lr, rho, epsilon)`

AdaMax is an Adam variation that uses the L^∞ norm, i.e. the absolutely maximum component of the gradient, eq. (18), it uses
This is implemented in the class `Adamax(lr, beta_1, beta_2, epsilon)`

Nadam is Adam with Nesterov update in the momentum part.
This is implemented in the class `Nadam(lr, beta_1, beta_2, epsilon)`

Optimizers: Initialization (I)

Initialization of weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ significantly affects the optimization process. What starting values should we choose?

- set them to zero, or one, or some constant (which one?)
- sample them from a Uniform or Normal distribution (what variance?)

The best choice is the one that allows information (signal) to flow forward and backward through a layer without chocking (squeezing) the signal.

- This happens when the variance of the inputs and outputs are close.

The above observation (Glorot and Bengio 2010) has led to various popular initialization schemes, which depend on the activation function used.

Activation	Uniform $(-r,\ r)$	Normal $(0,\sigma)$	a.k.a.
sigmoid	$r = \sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = \sqrt{\frac{2}{n_{in}+n_{out}}}$	Glorot
tanh	$r = 4\sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = 4\sqrt{\frac{2}{n_{in}+n_{out}}}$	
relu	$r = \sqrt{2}\sqrt{\frac{6}{n_{in}+n_{out}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{in}+n_{out}}}$	He

Table: Common initialization schemes

Optimizers: Initialization (II)

Given a layer, the number of inputs n_{in} is called **fan-in** and the number of outputs n_{out} is called **fan-out**.

In Keras, all initializers are in the the `keras.initializers` package.
Initializers are passed by name or instance to the constructor of the layer.
In a Dense layer we can use different initializers for the kernel, bias, and activation. Common initializers are:

- `Zeros()`, `Ones()`, `Constant(value)`
- `RandomUniform(minval, maxval, seed)`
  `RandomNormal(mean, stddev, seed)`
- `VarianceScaling(scale, mode='fan-in', distribution='normal', seed)`
  samples from normal with variance=scale/n where n=fan-in or n=fan-out
- `glorot_uniform(seed)`, `glorot_normal(seed)`
- `he_uniform(seed)`, `he_normal(seed)`
  sets $n_{out} = n_{in}$ and uses only fan-in
- custom: any function that takes a shape parameter and returns an ndarray

Fine-tuning: Decisions

The enormous flexibility of ANNs makes their tuning quite challenging. Decisions need to be made regarding

- topology: how deep, how many nodes per layer, what activations
- training: optimizers, initializers, regularizers
- other concerns such as:

1. Data normalization. It has been suggested that normalizing the mini-batch of inputs before they enter *each* layer yields more efficient training. This is called **Batch Normalization** (BN). In Keras this is done by using the class `keras.layers.BatchNormalization`.
2. **Learning rate scheduling**. A constant rate is unlikely to be optimal. A deterministic schedule can be used to decrease the learning rate as training proceeds. Popular choices are:
    - Piecewise constant: decrease the rate to a new constant after N steps
    - Exponential scheduling: decrease the rate by a constant factor every N steps
    - No explicit schedule: must use an adaptive optimizer to control the rate

Brute force grid search is unlikely to be useful with ANNs.

- We need to develop understanding and intuition about the nature of the problem and make targeted decisions.

Fine-tuning: Defaults

Practice has shown that the following can be good defaults to start.

Size: make the early layers wider and narrow them down near the output.
Depth: start shallow and gradually increase depth, based on performance.

Activation	ELU
Normalization	Batch
Initialization	He
Regularization	Dropout
Optimizer	Adam
LR Schedule	None

Table: Suggested initial choices for a dense ANN

ANN research is an empirical science (and art).
Experimentation and creativity are necessary to find optimal solutions.
It is also necessary to have good record keeping and reproducible research.

For details see the notebook **L11-DNNTuning-Examples.ipynb**

Further Reading

1. Read sections 7.1 and 7.12 in [GBC] "Regularization for Training Deep Models"
2. Read chapter 8 in [GBC] "Optimization for Training Deep Models"
3. Read the section on Faster Optimizers in chapter 11 of [GA] "Training Deep Neural Nets"

References

[GBC] Goodfellow I., Bengio Y., and Courville A. "Deep Learning". MIT, 2016

[GA] Géron Aurélien. "Hands-On Machine Learning with Scikit-Learn and TensorFlow". O'Reilly, 2017