Neural Networks: Introduction

Contents

Overview

...

Overview

Non-Linear Regression

A neural network is a form of supervised regression.

$f(x) = a_n W_n ... a_2 W_2 a_1 W_1 x$

with parameters W_n, and nonlinear operations a_n.

eg., a three-layer network:

$f(x; w_1, w_2, w_3 ...) = w_3 \cdot a(w_2 \cdot a(w_1 \cdot x))$

taking $a(z) = z$,

$f(x; w_1, w_2, w_3) = (w_3 w_2 w_1) \cdot x = w \cdot x$

ie., linear regression.

The network is an abstract description of how the regression model is composed. A network diagram describes inputs, weights, outputs via summations, and nonlinear operations (a).

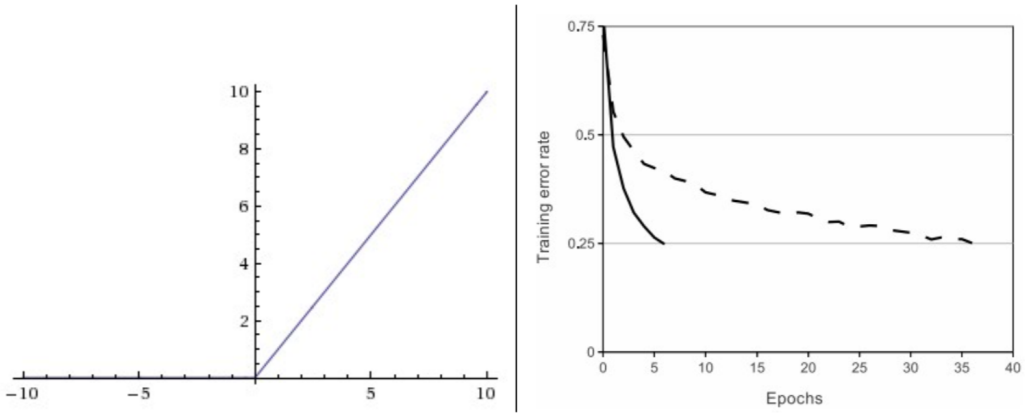
A network is a useful description when the regression model becomes extremely complex: parameterized by millions of weights; involving many nonlinear operations.

As a regression model, it can be used for classification via a decision function, which interprets real output as a score that is then mapped to a label. Classification networks are often terminated with a softmax operation which squishes abitarily-sized scores into a 0-to-1 range, permitting these to be interpreted as a probability of x being of label y.

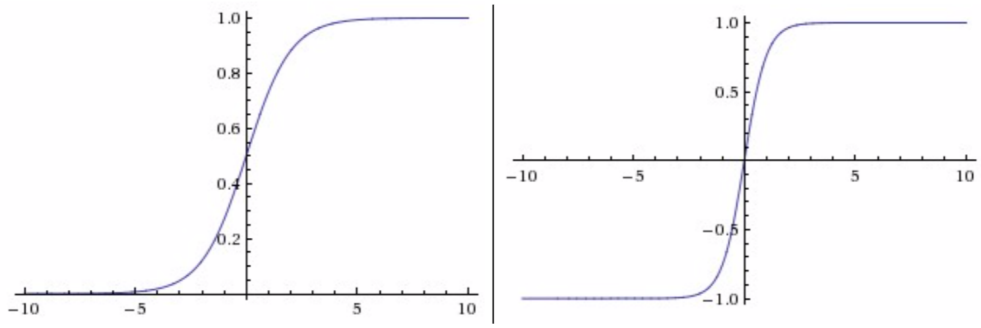
Network Diagrams

Activation Functions

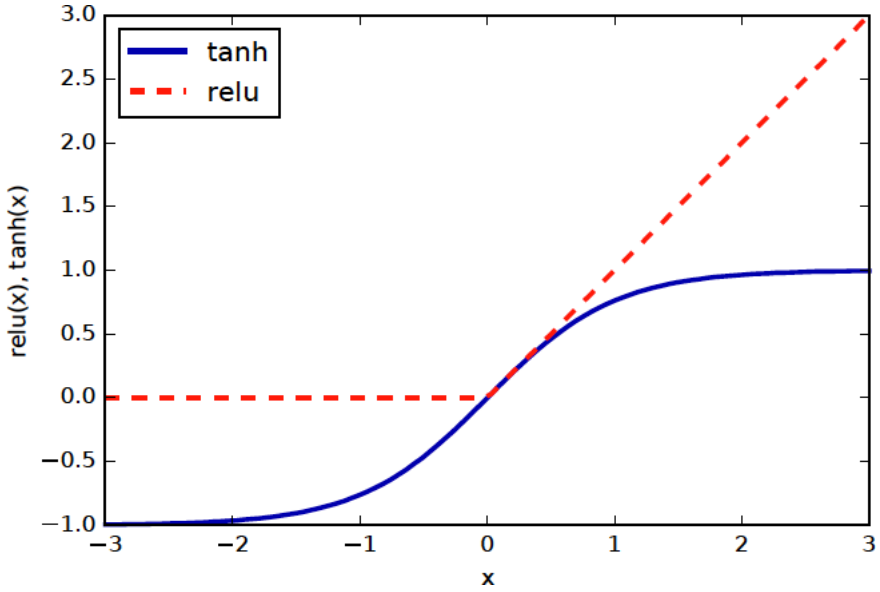
Example nonlinear functions include...



Left: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. **Right:** A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.



Left: Sigmoid non-linearity squashes real numbers to range between [0,1] **Right:** The tanh non-linearity squashes real numbers to range between [-1,1].



Forward and Backward Propogation

A neural network is trained in two steps: a forward and backward propogation.

The forward step computes the score for x, ie., $\hat{f}(x; w)$; and the $loss(\hat{f}(x; w), y)$.

The backward step computes the gradient of the loss with respect to the weights, so those weights can be updated.

Computational Graphs

This is a different algorithm than simple linear regression however, as the highly complex regression model is hard to compute under variations in its parameters. Given 1m weights, computing the loss for a small change in one parameter at a time requires intractably large computing power.

Making backprop tractable is a key advantage of the network representation of the regression model: a computational graph is build which mirrors this representation. As the graph executes in the forwards step, it partially computes intermediate pieces of the model known as layers.

As these intermedia terms (layers) are computed, the intermediate loss gradient is also stored. Using the chain rule (roughly: that gradients of pieces can be multiplied to give gradients of wholes), later gradients are produced by recursively multiplying all earlier ones.

This phrases the problem of finding the gradient of a 1m+ parameter function as a kind of dynamic programming: ie., start with the base case of the partial gradient of the output layer with respect to itself (always 1), and recurse multiplying the partial gradients of each prior layer until you hit the input. Since these were efficiently calculated on the forward pass, the backward pass is much cheaper than a naive gradient descent on f which varies the all paramters at once to find the gradient.

The weights are updated as the backprop proceeds, with the relevant loss gradient for that weight providing the update step.

A computational graph framework (eg., tensorflow, pytorch, etc.) is there essentially just a framework for tracking gradients. The purpose of computing a function f(x;w) with a graph, is so that as the computation proceeds (forwards), intermediate gradients are stored; you are then given a backwards pass "for-free".

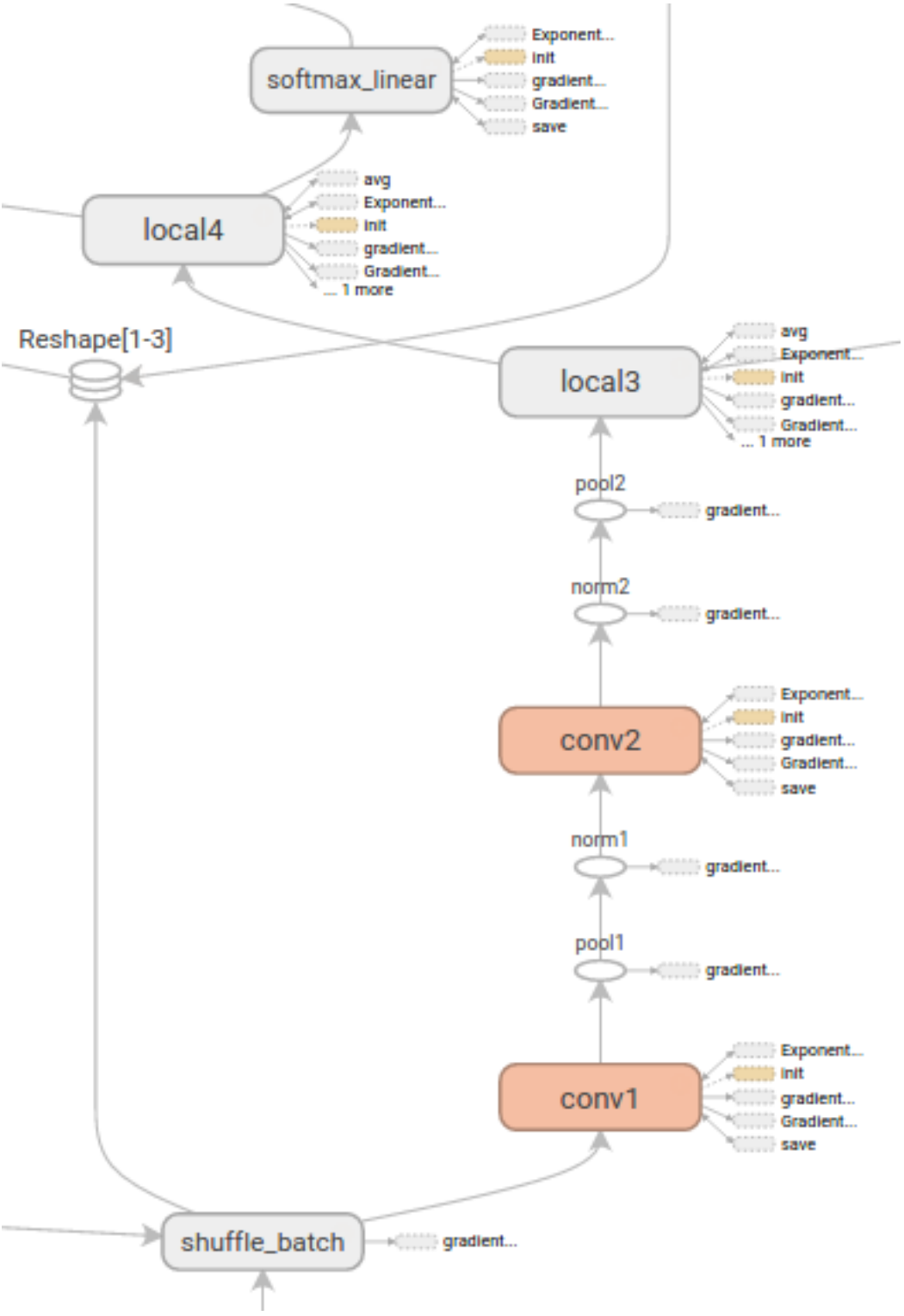


Figure 1: Partial Computational Graph

Neural Networks in Python

- Rough Sketch of a Forward Pass:

```
a = lambda x: 1.0/(1 + np.exp(-x))
x = np.array([0.15, 0.31, 0.28])
y = 1

w1, w2, b1, b2 = np.random.randn(4, 1)
h1 = a(w1 @ x + b1)
f = w2 @ h1 + b2

yp = int(f > 0)
l = (yp - y)**2
```

- Rough Sketch of Forwards/Backwards:

```
a = lambda : ... # eg., sigmoid
da = lambda : ... # eg. (1 - a)a , ie., derivative of sigmoid
class NeuralNetwork:
    def __init__(self, x, y):
        self.w1    = np.random.rand(self.input.shape[1],4)
        self.w2    = np.random.rand(4,1)
        self.y           = y

        self.layer0      = x
        self.layer1      = None
        self.layer2      = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = a(self.layer0 @ self.w1)
        self.layer2 = a(self.layer1 @ self.w2)

    def backprop(self):
        # application of the chain rule to find derivative of the loss function with respect to weights2 a
        dloss = 2*(self.y - self.layer2)
        dad2 = da(self.layer2)
        dad1 = da(self.layer1)

        dw2 = self.layer1 @ (dad2 * dloss)
        dw1 = self.layer0 @ (dad1 * (dad2 * dloss) @ self.w2))

        # update the weights with the derivative (slope) of the loss function
        self.w1 += 0.01 * dw1
        self.w2 += 0.01 * dw2
```

```
model = Sequential([
    Dense(512, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax'),
])
model.compile("adam", "categorical_crossentropy",
              metrics=['accuracy'])
```

Gradient Calculation with Graphs

Dense Networks and Alternatives

So far we have looked at dense networks, those where the layers are fully-connected to each other via weights.

For an image classification example, this corresponds to treating the input image, x, as everywhere affecting, every weight! Every reigon of x may cause an update in any weight, so that the weights are liable to be sensitive, eg. to the rotation of x. Since objects (eg., Cats, Knives, Guns, ..) are the same under rotation in space, dense networks often poorly capture this rotation-invarience.

Partially connecting regions of the input allows the network to capture the reccurent of the same pattern in different regions of the image, and may reduce sensitive to rotation and translation.

Performance, Training and Regularization

Each epoch (one run through the dataset), a neural network will get closer to perfect accuracy on the training data. With a network of billions of parameters, one should expect a large number of epochs to yield a network abitarily close to perfect: why? With enough capacity to extract information from data, the network will tend towards simply remembering what it has seen and therefore achiving perfect accuracy trivlally.

A model's performance is ultimately judged on the test set, not the training set. Since we should only use this once as a genuine, final estimation of model performance, we use a validation set to approximate test-set performance.

A neural network is trained to its optimal state when the validation pefromance is at its highest, which will be someways before its training performance peaks.

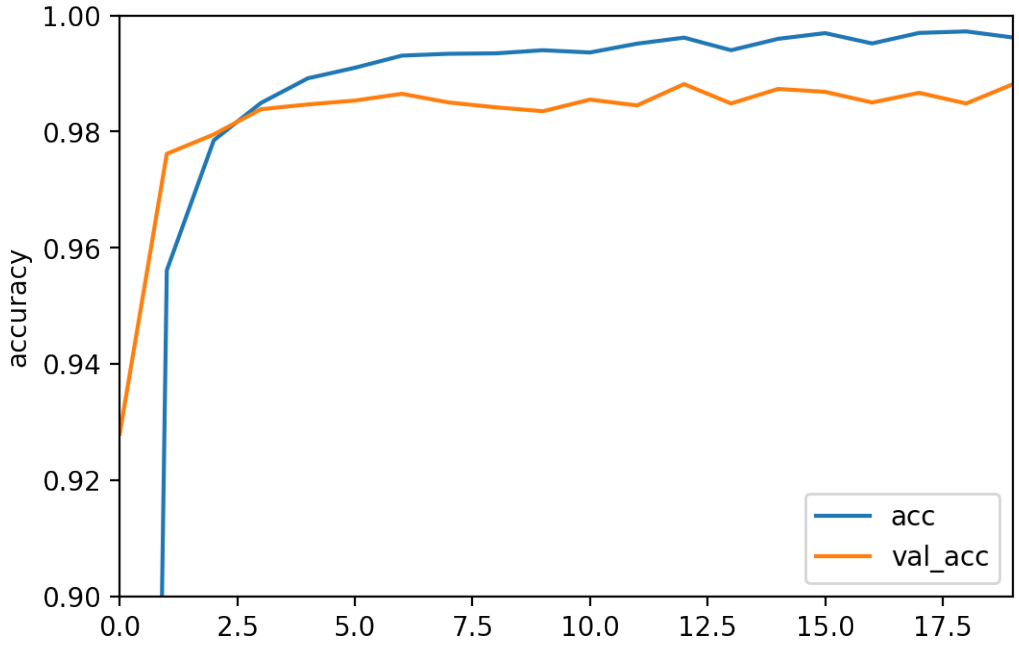


Figure 2: Training vs. Validation Loss

Observing Learning

Various techniques may be used to raise this performance and therefore the quality over the overall model. Firstly there are technquies such as data augmentation which increase performance without changing the model by supplying generated data, and then there's regularization which constrains the model at training-time to boost its performance at test-time.

In this section we'll consider issues with model performance that may be fixed by changes to the model (ie., essentially regularization technquies).

The validation-loss graphs should be observed during training. Stop training when overfitting is clearly observed. Retry with alternative hyperparameters until achiving best (tolerable) model performance.

Activation Functions

The choice of activation function has a regularizing effect. The more detail an activation function can capture, the greater risk we overfit during training.

Activation functions which discard information can therefore be helpful in reducing overfitting.

ReLU (rectified linear unit) is one such example.

Vanishing Gradients

The historical standard sigmoidal activation is hard to train with. As activation inputs grow (absolutely) large the activation output tends flat, and therefore the gradient tends to zero.

This tending to zero is known as vanishing. Since gradients are used to guide weight updates, a vanishing gradient for a weight renders that weight dead or untrainable. Even with well-chosen activation functions many weights in a network can be dead, reducing the network's ability to learn.

It is for this reason the LeakyReLU has been recently increasingly used, as it still has a gradient for negative inputs.

Weight Initialization

The choice of distribution to draw the weights from affects learning performance. Since weights are passed through multiple activation functions, there's a tendancy for the initial distribution to be sequeased around and deep-layer activations to be close to zero – which makes the network untrainable. Libraries such as tensorflow will choose reasonable inititalization strategies, but this is an area of active research.

Batch Normalization

One solution to the problem of activations tending to zero on repeated evaluations (ie., in deep layers) is simply to renormalize after every layer: to force that layer's mean to 0 and standard deviation to 1.

This reduces the dependenence on weight initialization strategy, and should allow higher learning rates (and therefore faster training).

Since the mean and varience of minibatches is used (ie., it is not the mean of the entire training set), this introduces some stochasticity in the network, which tends to regularize. This is only during training, as stochastic behaviour would lead to a non-deterministic prediction function; at test-time, the BN-layer uses a running empirical mean (, varience).

In addition to renormalization, BN layers shift and scale the input. The degree of shifting and scaling are learnt parameters and allow the network to retune its activation distributions to be effective for learning. In a simple case these parameters are just the mean and varience of the entire dataset, in which case the BN layer does not shift or scale (given x is already centred/scaled).

Batch normalization is an effective strategy to improve the performance of a neural network and should be widely used.

Optimization Strategies

While gradient descent is a feature of most optimization algorithms used for neural networks, many differ in how they use gradients and the degree that the learning parameter affects their performance.

The Adam Optimizer uses several additional techniuques.

If we consider the gradient as a local velocity of ascent, then naive gradient descent is very slow. We can increase the peformance by introducing:

(1) a momentum term keeps track of the velocity along the path we traversed. If we are going quickly recenrtly, but the local gradient slows, we keep some of that historical pace.

(2) an adaptive learning rate changes the step size. The adagrad and rmsprop algorithms do this by dividing the learning rate by the historical sum of squared gradients: this tends to decrease the step size, by penalizaling steps in large directions.

(3) learning rate decay. Change the learning rate during training or after each epoch; by gradually reducing the learning rate, the algorithm is able to find smaller minima. Starting larger than needed and growing much smaller, each sweep through the dataset can find increasing better minima.

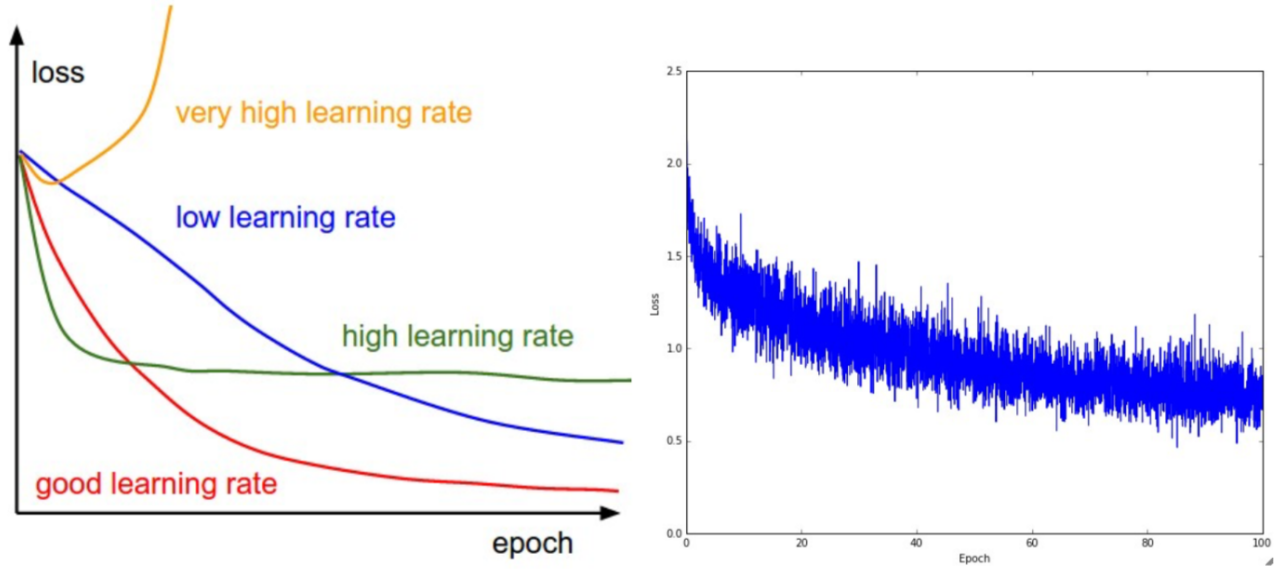
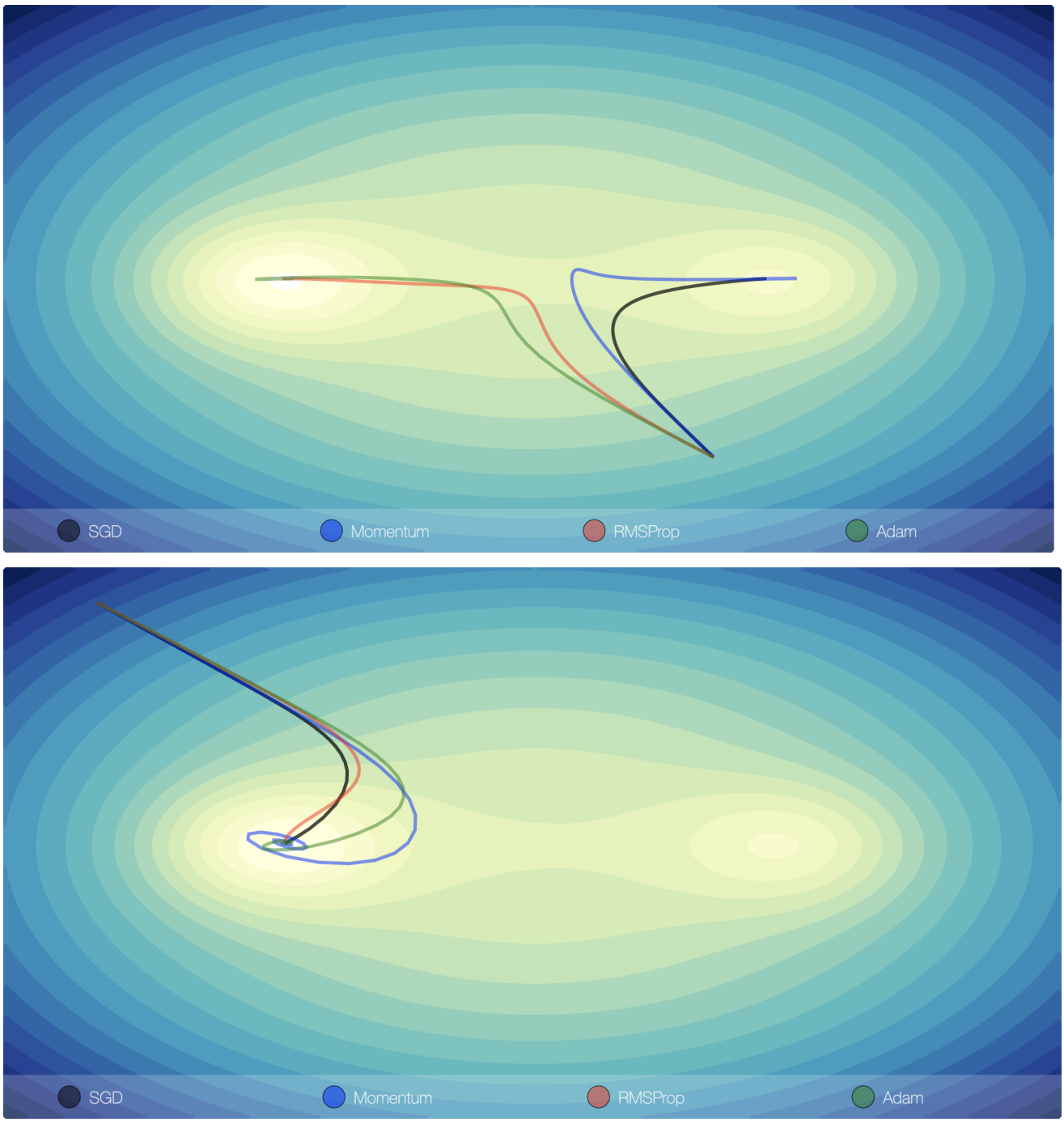


Figure 3: Learning Rate Tuning

Note however that small optimal minimal on the training set are unlikely to be optimial on the validation set. If a minimum loss can only be found by tiny permuations to the weights, it suggests this miniumuim is likely a noisey artefact of the training set.



cf. https://bl.ocks.org/EmilienDupont/raw/aaf429be5705b219aaaf8d691e27ca87/

Dropout

A very common neural network regularization strategy is dropout: randomly zero'ing activations in a layer.

Dropout is more common in dense layers. (In CNNs sometimes feature maps are dropped).

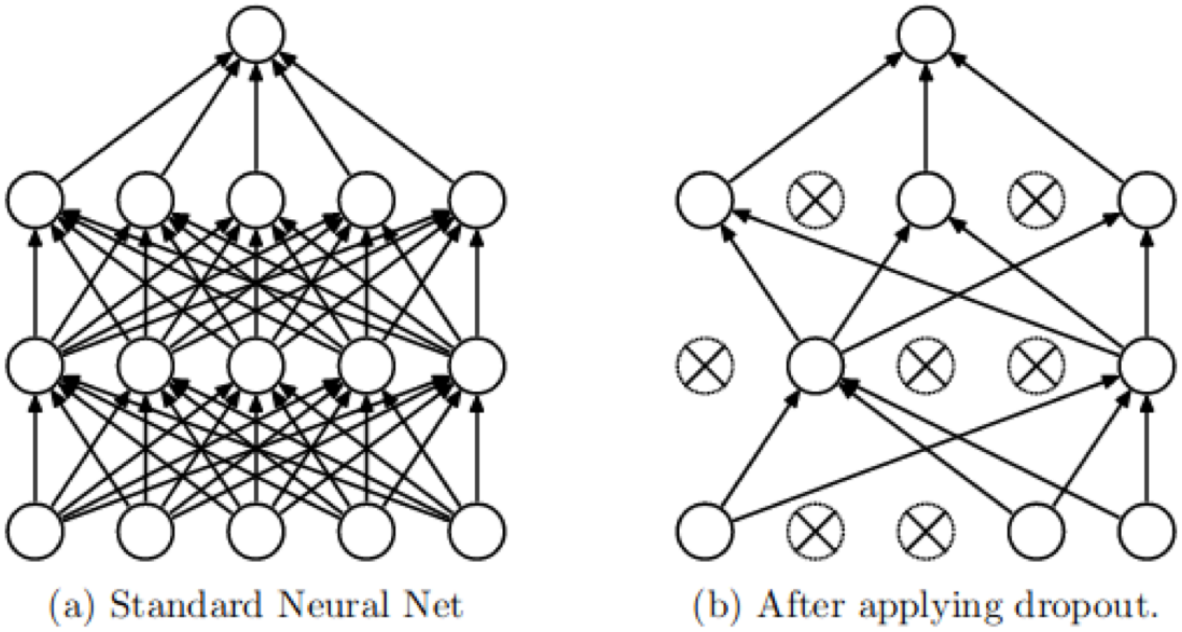


Figure 4: Dropout Pathways in a Neural Network

Why does dropout work?

Think of it as an ensemble of (partial) networks. Each is more focused on a specific region of input, and a specifc path from input to output. The target prediction is then an ensembling of these paths, with each better-trained on a part of the problem.

At training-time, dropout changes the function the network computes, from $\hat{y} = f(x; w)$ to $y' = f(x; w, d)$ where y' is a random variable because d, the dropout mask, is a random variable (of 0s, 1s specifying which activations are zero'd). This makes the prediction function non-deterministic, ie., that the same x may be classified differently.

So we do not use dropout at test time; the final value of all the weights calculated during training is part of the final model. However since these weights were learned under dropout, all the activations in the final model are scaled by the probability that they were dropped during training. Roughly that $\hat{y} = \hat{f}(x; w, p)$ where $d- > p$ by replacing each entry in d with the probability that this neuron was dropped during training.

General Training Methodology and Hyperparameter Search

1. Preprocess Data
2. Choose Architecture
3. Sanity Check A. Initialize and Run Once without regularization B. Check loss makes sense C. Add regularization and check loss D. Train without regularization to near-perfect performance
4. Tune learning rate, regularization parameters
5. Cross-Validate course->fine ranges A. Sample parameters over 10^1 with grid (, or random) layout B. Observe behaviour of loss, increase search space if bottoming-out C. Stop training if it explodes

In theory, a very small learning rate will find the best minimum; however the smaller the learning rate, the longer it takes to train.

Consider:

- Architecture
- Dropout Strength
- Learning Rate: $10^{-3}to10^{-5}$
- Small-sample grid search
- Validation vs training loss

[1] . . .

Regularization Strategies

As a final remark, note that often the strategy with regularizing a neural network is to introduce some randomness at training-time which is averaged-out at test-time.